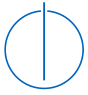


TUM School of Computation, Information and Technology
Technische Universität München

TUM

Function Delivery Network: Bringing Serverless Computing to Edge-Cloud Continuum

Anshul Jindal

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Michael G. Bader

Prüfer*innen der Dissertation:

1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Rajkumar Buyya
3. Prof. Thomas Fahringer, Ph.D.

Die Dissertation wurde am 04.10.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 20.06.2023 angenommen.

Zusammenfassung

Mit dem Aufkommen des Edge Computing werden die Berechnungen auf Edge-Geräte verlagert. Die endgültige Architektur ist eine hybride Umgebung, die Edge-Geräte mit der Cloud verbindet und das Edge-Cloud-Kontinuum bildet. Viele Anwendungen werden über das Kontinuum verteilt, um die Heterogenität zu nutzen. Die Programmierung und Bereitstellung dieser Anwendungen über das Kontinuum hinweg ist jedoch aufgrund der unterschiedlichen Rechen- und Datenanforderungen eine Herausforderung. Serverless Computing, ein Cloud-Computing-Modell, das Entscheidungen über die Infrastrukturverwaltung abstrahiert, scheint eine ideale Lösung für diese schwierigen Aufgaben zu sein. Function-as-a-Service (FaaS), eine wichtige Voraussetzung für Serverless Computing, ermöglicht die Zerlegung einer Anwendung in einfache, eigenständige Funktionen, die auf einer Serverless Compute Plattform (z. B. AWS Lambda) ausgeführt werden. Die Serverless Compute Platform ist für die Bereitstellung von Ressourcen für die Funktionen verantwortlich. Eine Serverless Compute Platform unterstützt jedoch keine nahtlose Funktionsbereitstellung über das gesamte Edge-Cloud-Kontinuum hinweg. Sie ist auf eine einzige Cloud beschränkt und unterstützt derzeit nur homogene Rechenknoten. Außerdem gibt es keine Möglichkeit, private Cluster neben öffentlichen Cloud-Clustern in eine Serverless Compute Platform einzubinden. Darüber hinaus gibt es zahlreiche serverlose Plattformen, die jeweils über einen eigenen Virtualisierungs- und Überwachungsstack verfügen.

Diese Dissertation adressiert die oben genannten Herausforderungen durch die Entwicklung einer Erweiterung des FaaS-Konzepts als Programmierschnittstelle für Serverless Computing über das Edge-Cloud-Kontinuum, Multi-Cloud und Hybrid-Cloud. Diese Erweiterung ist ein Netzwerk verteilter heterogener Serverless Compute Cluster, genannt Function Delivery Network (FDN). FDN integriert nahtlos die Edge- und Cloud-Cluster und ermöglicht es dem Benutzer, die Funktionen im Kontinuum einzusetzen und aufzurufen. Wir demonstrieren die Effektivität von FDN anhand von sechs Clustern, die auf vier Plattformen basieren: 1) OpenWhisk, 2) OpenFaaS, 3) AWS Lambda und 4) Google Cloud Functions (GCF), verteilt über das Edge, in der Cloud und on-premise. Um das Problem der Integration und Verwaltung von verteilten Clustern in FDN zu lösen, nutzen wir Virtual Kubelet, eine Open-Source-Kubernetes-Kubelet-Implementierung. FDN erstellt virtuelle Knoten, die die Cluster repräsentieren, und verwendet die Kubernetes-Knotenverwaltungsfunktionen, um sie zu verwalten. Für die Überwachung von Clustern, die auf heterogenen Plattformen basieren, haben wir außerdem den FDN-Monitor entwickelt. Er wird als Sidecar mit jedem virtuellen Knoten in FDN eingesetzt, um die Metrikdaten aus dem entsprechenden Cluster zu ziehen und sie in eine einheitliche Metrik umzuwandeln.

Um das Verhalten von FaaS-Funktionen im FDN zu charakterisieren, erstellen wir zwei Modelle: 1) Das Functions Performance Model, das verschiedene statistische Ansätze verwendet, und 2) das Functions Interaction Model, das neuronale Temporal Point Processes (TPPs) verwendet. Unsere Bewertung des Functions Performance Model zeigt relativ genaue Vorhersagen, mit einer Genauigkeit von mehr als 75% für AWS Lambda und GCF. Bei der Vorhersage des Funktionstyps in einer serverlosen Anwendung erreichte das Functions Interaction Model eine Genauigkeit von über 94%, und bei der Vorhersage der Funktionsaufrufzeit liegt der mittlere absolute Fehler unter 22ms.

Bei der Serverless Compute Platform haben die Endnutzer keine Kontrolle darüber, wo eine Funktion ausgeführt wird. Darüber hinaus sind für die Planung von Funktionsaufrufen über das Kontinuum hinweg Informationen über die Reaktionszeiten der Funktionsaufrufe sowie die Berechnungs- und Datenanforderungen erforderlich. Um dieses Problem zu lösen, haben wir Courier entwickelt. Courier liefert die Funktionsaufrufe an eine geeignete Teilmenge von Clustern im Kontinuum auf der Grundlage von Funktions- und Datenkenntnissen. Die Aufrufe werden mit Hilfe der beiden entwickelten Lastausgleichsalgorithmen auf die ausgewählte Untergruppe von Clustern verteilt: Latency-Aware und Service Level Objective (SLO)-Aware. Unsere Evaluierungsergebnisse zu verschiedenen Funktionsbenchmarks und einer serverlosen Anwendung zeigten, dass der SLO-Aware-Algorithmus am besten abschnitt und die P90-Antwortzeit der Funktion bei Verwendung dieses Algorithmus die definierte SLO einhielt.

Um das Problem zu lösen, die optimalen Speicherkonfigurationen für FaaS-Funktionen innerhalb einer serverlosen Anwendung zu finden, die die Kosten minimieren und das SLO erfüllen, haben wir eine externe Komponente für FDN namens SLAM: SLO-Aware Memory Optimization entwickelt. Wir demonstrieren SLAM auf AWS Lambda, und die Ergebnisse zeigen, dass die vorgeschlagenen Speicherkonfigurationen garantieren, dass mehr als 95% der Anforderungen innerhalb der definierten SLOs abgeschlossen werden. Darüber hinaus haben wir zwei Algorithmen zur Erkennung von Anomalien entwickelt, um die Zuverlässigkeit der Cluster im FDN zu gewährleisten: 1) Online-Erkennung von Speicher-lecks mithilfe von Precog und 2) Erkennung von anomalen virtuellen Maschinenmonitoren mithilfe von IAD: Indirect Anomaly Detection. Die Leistungsbewertung zeigte, dass Precog einen F1-Score von 0.85 mit weniger als einer halben Sekunde Vorhersagezeit auf den realen Workloads erreichen kann. Die Leistungsbewertung des IAD-Algorithmus auf vier Datensätzen zeigt, dass er eine durchschnittliche Genauigkeit von 83,7% erreichen kann.

Abstract

With the emergence of edge computing, computation is being pushed towards edge devices. The final architecture is a hybrid environment, connecting edge devices to the cloud and forming the edge-cloud continuum. Many applications are distributed over the continuum to leverage heterogeneity. However, programming and deploying these applications across the continuum is challenging due to the varying compute and data requirements. Serverless computing, a cloud computing model that abstracts infrastructure management decisions, appears to be an ideal solution for solving these challenging tasks. Function-as-a-Service (FaaS), a key enabler of serverless computing, allows an application to be decomposed into simple, standalone functions executed on a serverless compute platform (e.g., AWS Lambda). The serverless compute platform is responsible for deploying and facilitating resources to the functions. However, serverless compute platforms do not support seamless function deployments across the edge-cloud continuum. They are confined to a single cloud and currently only support homogeneous compute nodes. Furthermore, there is no provision to incorporate private clusters alongside public cloud clusters in serverless compute platforms. Moreover, numerous serverless platforms exist, each with a distinct virtualization and monitoring stack.

This dissertation addresses the above challenges by developing an extension to the concept of FaaS as a programming interface for serverless computing across the edge-cloud continuum, multi-cloud and hybrid-cloud. This extension is a network of distributed, heterogeneous serverless compute clusters called Function Delivery Network (FDN). FDN seamlessly integrates the edge and cloud clusters and allows the user to deploy and invoke the functions in the continuum. We demonstrate the effectiveness of FDN using six clusters based on four platforms: 1) OpenWhisk, 2) OpenFaaS, 3) AWS Lambda, and 4) Google Cloud Functions (GCF), distributed across the edge, in the cloud, and on-premise. To address the problem of integrating and managing distributed clusters in FDN, we leverage Virtual Kubelet, an open-source Kubernetes kubelet implementation. FDN creates virtual nodes representing the clusters and uses Kubernetes node management capabilities to manage them. Furthermore, for monitoring clusters based on heterogeneous platforms, we created FDN-Monitor. It is deployed as a sidecar with every virtual node in FDN to pull the metrics data from the corresponding cluster and converts them to a unified metric.

To characterize the behavior of FaaS functions in FDN, we create two models: 1) Functions Performance Model using various statistical approaches and 2) Functions Interaction Model using neural Temporal Point Processes (TPPs). Our evaluation of the Functions Performance Model shows relatively accurate predictions, with an accuracy greater than 75% for AWS Lambda and GCF. For function type prediction in a serverless application, the Functions Interaction Model achieved an accuracy of over 94%, and for function invocation time prediction, the mean absolute error is below 22ms.

Serverless compute platforms do not allow end-users control over where a function is executed. Furthermore, scheduling function invocations across the continuum require information on the function's invocations response times and computation and data requirements. To tackle this problem, we developed Courier. Courier delivers the function invocations to a suitable subset of clusters in the continuum based on function awareness and data awareness. The invocations are load balanced across the selected subset of clusters using

the two developed load balancing algorithms: Latency-Aware, and Service Level Objective (SLO)-Aware. Our evaluation results on different function benchmarks and a serverless application showed that the SLO-Aware algorithm performed the best, and the function's P90 response time when using it, adhered to the defined SLO.

To address the problem of finding the optimal memory configurations for FaaS functions within a serverless application that minimizes cost and meets SLO, we developed an external component to FDN called SLAM: SLO-Aware Memory Optimization. We demonstrate SLAM on AWS Lambda, and the results show that the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs. Furthermore, to provide reliability across the clusters within FDN, we developed two anomaly detection algorithms: 1) Online memory leak detection using Precog, and 2) Anomalous Virtual Machine Monitors detection using IAD: Indirect Anomaly Detection. The performance evaluation showed that the Precog can achieve a F1-Score of 0.85 with less than half a second prediction time on the real workloads. The performance evaluation of the IAD algorithm on four datasets shows that it can achieve an average accuracy score of 83.7%.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Michael Gerndt. It all started with him providing me with an opportunity to do a student job during my master's studies, and then offering me to pursue the Ph.D. degree. I would like to thank him for all his invaluable advice, support, and help during my Ph.D. study and work at the university. Next, I would like to thank Prof. Shajulin Benedict for connecting me with Prof. Gerndt and for all our collaborations. I would also like to thank Vladimir, with whom my research journey started from guided research to many collaborations. Thank you for offering me continuous advice and encouragement throughout those collaborations. I would also like to thank Mohak for our long discussions in the office, for reviewing my papers, and for our past collaborations.

Additionally, I would like to extend my sincere thanks to Prof. Jorge Cardoso, my Ph.D. mentor, for supporting and encouraging me during the internship at Huawei and in our past collaborations from software campus. I would also like to thank other colleagues at Huawei for the collaborations, discussions, and support, especially Apoorv, Vittorio, Paul, and Ilya.

I would like to thank my colleagues, former and current, at the Technical University of Munich, CAPS chair, for the discussions and support in the past four years, especially Prof. Schulz, Amir, Jophin, Andreas, Fariz, Roman, Dai Yang, Lisa, Bengisu, Dai Liu, Isaías, Eishi, Jianfeng, and Paolo. My special thanks to Jürgen for providing all the hardware equipments.

Throughout my stay at TUM, I was lucky to work with bright students. Thank you, Marko, Lennart, Thomas, Muthuraman, Chen, Lucas, Julian, Stephan, Gor, Gurudeep, Raj, Tetiana, Riccardo, Markus, Astghik, Christopher, Joshua, Max, Hady, and Michael Lohr.

Getting through my dissertation required more than academic support, and this journey would have been impossible without the support of my friends. I want to thank Nishant, Aneesha, Siddharth, Abhishek, and Vatsala. A special thanks to Merve for supporting me and helping me review this dissertation.

I am also thankful to the German Federal Ministry of Education and Research (BMBF) for selecting me and funding my Software Campus project (BEHAVE). Additionally, I would like to thank Google Cloud for providing Google Cloud Platform research credits, which helped a lot during my research.

Last, but most importantly, none of this could have happened without my family and parents. I would like to thank my parents for their unconditional support.

Contents

Zusammenfassung iii

Abstract v

Acknowledgments vii

1 Introduction 1
1.1 Motivation . . . 2
1.2 Dissertation Contributions . . . 4
1.3 Organization of the Dissertation . . . 6

2 Background 8
2.1 Virtualization: a Cloud-Enabling Technology . . . 8
2.1.1 Hardware-level virtualization . . . 8
2.1.1.1 CPU Virtualization . . . 9
2.1.1.2 Memory Virtualization . . . 9
2.1.1.3 I/O Virtualization . . . 10
2.1.2 OS-level virtualization . . . 10
2.2 Edge-to-Cloud Continuum . . . 11
2.2.1 Cloud Computing . . . 12
2.2.1.1 Cloud Deployment Models . . . 13
2.2.1.2 Cloud Service Models . . . 13
2.2.2 Edge Computing . . . 14
2.3 Serverless Computing . . . 15
2.3.1 FaaS Function Invocation Procedure . . . 16
2.3.1.1 Cold-start Problem . . . 17
2.3.2 Serverless Compute Platforms . . . 18
2.3.2.1 OpenWhisk . . . 18
2.3.2.2 OpenFaaS . . . 19
2.3.2.3 Google Cloud Functions (GCFs) . . . 19
2.3.2.4 AWS Lambda . . . 20
2.4 Cloud Application's Architectures . . . 20
2.4.1 Monolithic Application Architecture . . . 20
2.4.2 Microservices Application Architecture . . . 21
2.4.3 FaaS-based Application Architecture . . . 22

3 Related Work 23
3.1 Microservices vs Serverless Applications . . . 23
3.2 Heterogeneity in the FaaS Workloads . . . 24

3.3 Heterogeneity among the Serverless Compute Platforms . . . 25
3.4 Serverless Computing across the Edge-Cloud Continuum . . . 26
3.5 Serverless Computing across Multi-Cloud . . . 27
3.5.1 Solutions Connecting Multiple Cloud Platforms . . . 27
3.5.2 Solutions Connecting Multiple Serverless Compute Platforms . . . 27
3.6 Data-Aware Scheduling in Serverless Computing . . . 28
3.7 Memory Optimization of Serverless Applications . . . 29

4 FDN: Function Delivery Network **30**
4.1 FDN Design Overview . . . 30
4.1.1 Requirements . . . 31
4.1.1.1 Functional Requirements . . . 31
4.1.1.2 Non-Functional Requirements . . . 32
4.1.2 Design Methodology . . . 32
4.1.3 FDN High-level Architecture . . . 34
4.2 FDN Components . . . 35
4.2.1 FDN's Serverless Compute Clusters . . . 35
4.2.1.1 Cluster Types . . . 36
4.2.1.2 Clusters Creation Automation . . . 37
4.2.2 FDN-Monitor . . . 38
4.2.3 FDN Inventory Database . . . 41
4.2.4 Clusters Management . . . 43
4.2.5 Data Orchestrator . . . 44
4.2.6 Functions Management . . . 46
4.2.7 Behave . . . 47
4.2.8 Courier Control Plane and Load Balancer . . . 48
4.2.9 FDN-UI . . . 49
4.3 Summary . . . 49

5 Behave: Behavioral Modeling of FaaS Functions in FDN **50**
5.1 Functions Performance Model . . . 50
5.1.1 Function Capacity (FC) . . . 50
5.1.2 FnCapacitor . . . 51
5.1.3 Experimental Configuration . . . 53
5.1.3.1 Monitoring Metrics . . . 54
5.1.4 Experimental Results . . . 54
5.1.4.1 Memory Effect on Function Execution Duration . . . 54
5.1.4.2 Memory Effect on Function's Concurrent Instances . . . 55
5.1.4.3 Effect of Function Concurrency on the FC . . . 56
5.1.4.4 Function Capacity Estimation . . . 56
5.2 Functions Interaction Model . . . 58
5.2.1 Temporal Point Processes (TPPs) . . . 58
5.2.1.1 Neural Temporal Point Processes Models . . . 61
5.2.2 TppFaaS - Developed System . . . 63
5.2.2.1 Sampler . . . 64
5.2.2.2 TPP Models . . . 66
5.2.3 Evaluation Settings . . . 67
5.2.3.1 Benchmark Applications . . . 67

5.2.3.2 Infrastructure Settings . . . 68
5.2.3.3 Dataset Generation . . . 68
5.2.3.4 Training Details and Model Parameters . . . 68
5.2.3.5 Performance Quality Measures . . . 69
5.2.4 Results . . . 70
5.2.4.1 Predictions on Datasets without Cold Starts . . . 70
5.2.4.2 Prediction on Datasets with Cold Starts . . . 73
5.3 Summary . . . 75

6 Courier: Users's Functions Invocations Delivering and Load Balancing in FDN 76
6.1 Introduction . . . 76
6.2 Courier Load Balancer . . . 78
6.2.1 Courier Load Balancer Configuration . . . 79
6.3 Courier Control Plane . . . 81
6.3.1 Function Delivery Policies . . . 81
6.3.1.1 Function-Aware Delivery Policy . . . 81
6.3.1.2 Data-Aware Delivery Policy . . . 82
6.3.2 Load Balancing Algorithms . . . 83
6.3.2.1 Latency-Aware Load Balancing Algorithm . . . 84
6.3.2.2 SLO-Aware Load Balancing Algorithm . . . 85
6.3.3 Load Balancer Configurator . . . 85
6.4 Summary . . . 86

7 SLAM: SLO-Aware Memory Optimization of Serverless Applications in FDN 87
7.1 Introduction . . . 87
7.2 SLAM Tool . . . 89
7.2.1 Load Generator . . . 90
7.2.2 Application Call Graph Builder . . . 90
7.2.3 Functions Performance Modeler . . . 91
7.2.4 Application Execution Time Estimator . . . 91
7.2.5 Config Finder . . . 92
7.2.5.1 Optimization Objectives . . . 92
7.2.5.2 Optimal Memory Configuration Finding Algorithm . . . 93
7.3 SLAM Evaluation . . . 95
7.3.1 Evaluation Settings . . . 95
7.3.1.1 Test Applications . . . 95
7.3.1.2 Evaluation Questions . . . 95
7.3.2 Results . . . 96
7.3.2.1 Q1. SLAM Estimation Time Accuracy . . . 96
7.3.2.2 Q2. SLAM Configuration Finding Accuracy . . . 98
7.3.2.3 Q3. SLAM Configuration Finding Efficiency and Scalability . . . 100
7.4 Summary . . . 101

8 Anomaly Detection in the FDN 102
8.1 Online Memory Leak Detection . . . 102
8.1.1 Methodology for Memory Leak Detection . . . 103
8.1.1.1 Problem Statement . . . 103
8.1.1.2 Illustrative Example . . . 104

8.1.2 Memory Leak Detection Algorithm: Precog . . . 104
8.1.3 Precog Evaluation . . . 106
8.1.3.1 Q1. Memory Leak Detection Accuracy . . . 106
8.1.3.2 Q2. Scalability . . . 107
8.1.3.3 Q3. Parameter Sensitivity . . . 108
8.2 Anomalous VMMs Detection . . . 109
8.2.1 Problem Definition . . . 109
8.2.1.1 Illustrative Example . . . 110
8.2.2 Indirect Anomaly Detection (IAD) Algorithm . . . 111
8.2.2.1 IAD Algorithm . . . 111
8.2.2.2 Test Module . . . 112
8.2.3 Experimental Settings . . . 113
8.2.3.1 Datasets . . . 113
8.2.3.2 Evaluated Algorithms . . . 114
8.2.3.3 Other Settings . . . 115
8.2.4 Results . . . 115
8.2.4.1 Q1. Indirect Anomaly Detection Accuracy . . . 115
8.2.4.2 Q2. Anomalous VMMs Finding Efficiency and Scalability . . . 116
8.3 Summary . . . 117

9 Function Delivery Network Evaluation Settings **118**
9.1 Benchmarks . . . 118
9.1.1 FaaS Functions . . . 119
9.1.1.1 Web-based FaaS Functions . . . 119
9.1.1.2 CPU-Intensive FaaS Functions . . . 119
9.1.1.3 Memory-Intensive and Disk I/O-Intensive FaaS Functions . . . 119
9.1.1.4 Network I/O-Intensive FaaS Functions . . . 120
9.1.1.5 ML-based FaaS Functions . . . 120
9.1.2 Serverless Application . . . 121
9.2 Heterogeneous Target Serverless Compute Clusters . . . 122
9.2.1 Edge-Clusters . . . 122
9.2.2 Cloud-Clusters . . . 123
9.2.2.1 Private-Cloud-Clusters . . . 123
9.2.2.2 Public-Cloud-Clusters . . . 124
9.3 Evaluation Infrastructure . . . 125
9.3.1 FDN Deployment Settings . . . 125
9.3.2 FDN Test Framework . . . 126
9.3.2.1 Configuration File . . . 127
9.3.2.2 FDN Test Client . . . 127
9.3.2.3 FDN Load Generator . . . 127
9.4 Performance Quality Metrics . . . 129
9.4.1 User-Centric Metrics . . . 130
9.4.2 Platform-Centric Metrics . . . 130
9.5 Summary . . . 131

10 Function Delivery Network Evaluation Results **132**
10.1 FaaS Functions Performance and Resources Usage . . . 132
10.1.1 Web-based Function . . . 133

10.1.1.1 nodeinfo . . . 133
10.1.2 CPU-Intensive Functions . . . 135
10.1.2.1 primes . . . 135
10.1.2.2 linpack . . . 136
10.1.2.3 sentiment-analysis . . . 138
10.1.3 Memory- and Disk-Intensive Functions . . . 140
10.1.3.1 dd . . . 140
10.1.3.2 gzip-compression . . . 141
10.1.4 Network-Intensive Function . . . 143
10.1.4.1 json-loads . . . 143
10.1.5 ML-Based Functions . . . 144
10.1.5.1 lr-prediction . . . 144
10.1.5.2 image-processing . . . 146
10.2 FaaS Functions Performance and Resources Usage Summary . . . 147
10.3 FDN's Performance Overhead . . . 150
10.4 FDN's Function Delivery Policies Correctness . . . 151
10.5 FDN's Bucket Replication Performance . . . 154
10.6 FDN's Load Balancing Algorithms Performance . . . 155
10.6.1 Individual FaaS Function (nodeinfo) . . . 156
10.6.1.1 Performance on Low Workload (Trace R2) . . . 156
10.6.1.2 Performance on High Workload (Trace R1) . . . 160
10.6.2 Individual FaaS Function (gzip-compression) . . . 162
10.6.2.1 Performance on Low Workload (Trace R2) . . . 162
10.6.2.2 Performance on High Workload (Trace R1) . . . 166
10.6.3 Individual FaaS Function (lr-prediction) . . . 168
10.6.3.1 Performance on Low Workload (Trace R2) . . . 168
10.6.3.2 Performance on High Workload (Trace R1) . . . 170
10.6.4 Serverless Application (faas-composer) . . . 171
10.6.4.1 Performance on Low Workload (lowered-down version of Trace R1) . . . 172
10.7 FDN's Load Balancing Performance Summary . . . 178
10.7.1 Algorithms Performance . . . 179
10.7.2 Clusters Performance . . . 181
10.8 Summary . . . 181

11 Conclusion and Future Outlook **183**
11.1 Conclusion . . . 183
11.2 Future Outlook . . . 185
11.2.1 Extension of Virtual Kubelet . . . 185
11.2.2 Energy Efficiency and Power-aware Scheduling on Edge Clusters . . . 186
11.2.3 Improvement of the Function Delivering Decision-making Policies . . . 186
11.2.4 Improvement of the Load-Balancing Algorithms . . . 186
11.2.5 Shim for Function Code for all the Serverless Compute Platforms . . . 187
11.2.6 Creating Serverless Storage Backends for the FDN . . . 187
11.2.7 Distributed Anomaly Detection in the FDN . . . 187

Appendices **188**

Appendix A Function Delivery Network Configurations **189**

A.1 FDN Design Configurations Templates 189
A.1.1 FDN-Provider Deployment Template 189
A.1.2 FDN-Provider Function Deployment Template 191
A.2 FDN-Components 193
A.2.1 FDN-Monitor Grafana Dashboards 193
A.2.2 FDN-UI 194
A.3 FDN Test Framework 194

Appendix B Source Code Availability **196**

Appendix C List of Authored and Co-authored Publications **197**
C.1 Publications Associated with the Dissertation 197
C.1.1 Journal Articles 197
C.1.2 Conference Articles 197
C.1.3 Workshop Articles 198
C.1.4 Poster 198
C.2 Other Publications 198
C.2.1 Journal Articles 198
C.2.2 Conference Articles 199
C.2.3 Workshop Articles 199

Index **200**

List of Figures **203**

List of Tables **208**

List of Algorithms **210**

List of Listings **211**

Acronyms **212**

Bibliography **215**

Webliography **228**

1

Introduction

“The only way to do great work is to love what you do."

— Steve Jobs

With the emergence of cloud computing, the data is transferred to the cloud data centers through the network for computation and storage [62, 203]. However, with more and more Internet of Things (IoT) devices generating data, edge computing emerged, where computation is being pushed towards edge devices [61]. On the one hand, the data is processed close to its source, decreasing the latency [91]. On the other hand, edge devices are usually limited in resources, limiting their compute power compared to the cloud resources. The final architecture is a hybrid environment, connecting edge devices to the cloud and forming an edge-cloud continuum [48]. Many of today’s applications are spread over the edge-cloud continuum [246]. (1) Web applications, for instance, combine mobile devices, edge computers for content delivery, and servers to enable interaction and collaboration. (2) IoT applications use microcontrollers, mini-computers, edge computers, and servers for delivering sensor measurements and controlling devices in the physical world. (3) Large-scale experiments gather big data sets that need to be preprocessed and aggregated, forwarded to analytics functions, fed into compute-intensive simulations, and visualized for the scientists. These applications are highly dynamic with respect to their structure and workload [246]. *Programming and deploying these applications is a highly challenging task. This is due to the heterogeneity of the underlying hardware, varying compute, and data access requirements across time and application components, as well as the dynamic structure of the applications due to agile programming techniques combined with continuous delivery.*

Serverless computing is a cloud computing model that abstracts server management and infrastructure decisions away from the users [304]. Significant progress has been made in the context of cloud computing based on the idea of *serverless computing* since its launch in November 2014 [39]. In this model, the allocation of resources is managed by the cloud service provider rather than by the team of application developers and deployment managers, i.e., *DevOps*, thereby increasing their productivity. Function-as-a-Service (FaaS) is a key enabler of serverless computing [304]. In FaaS, a serverless application is decomposed into simple, standalone functions that are uploaded to a serverless compute platform for execution. These functions are stateless, i.e., the state is not kept across function invocations. Functions can be invoked by a user’s

HTTP request or by another type of event created within the serverless compute platform. The serverless compute platform is responsible for providing resources for function invocations and performs automatic scaling. This is done by creating an execution environment that provides a secure and isolated runtime for the function. Currently, a significant number of open source and commercial serverless compute platform are available [204]. AWS Lambda [260], Azure Functions [194] and Google Cloud Functions [116] are few examples of commercial serverless compute platform, and OpenWhisk [225] and OpenFaaS [217] are two examples of open-source serverless compute platform. OpenWhisk is also the platform that leverages IBM's FaaS offering IBM Cloud Functions [134]. The amount of resources for an execution environment is typically decided based on the maximum amount of memory and execution time (timeout) statically specified by the user on function creation [127]. The amount of memory configured is important, since some commercial serverless compute platform providers increase the amount of compute available to the function when more memory is assigned [156, 71]. If a function invocation violates these constraints, the serverless compute platform immediately terminates the invocation [321, 97, 253]. Therefore, a function invocation might get prematurely terminated if it requires high computing power and is executed in an execution environment with low compute capabilities. Additionally, in the last couple of years, there has been a shift observed in the cloud native applications' architecture from independently deployable microservices towards serverless architecture, which is more decentralized and distributed [162]. From an economic point of view, this deployment model can reduce the cost of operation due to fine-grained on-demand automatic scaling. Additionally, the lack of server management can decrease the time-to-market for an application [249]. The serverless computing paradigm can be used for building a myriad of applications such as web applications, IoT, BigData workloads, Chatbots and Amazon Alexa, as well as IT Automation [109, 72].

1.1 Motivation

Serverless computing seems to be the perfect solution for solving the problems faced by the users for deploying the applications across the edge-cloud continuum. However, despite having many advantages, usage of serverless computing across the edge-cloud continuum suffers from the following pain points:

Public cloud serverless compute platforms are limited to homogenous nodes: All the public cloud provider's serverless compute platforms are limited to clusters of homogeneous nodes, i.e., the underneath node's CPU architectures such as x86_64/amd64 and arm64v8 remain fixed [71, 263]. Furthermore, the nodes that are part of the cluster do not support the CPU acceleration devices such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs). As a result, functions that are heavily based on video processing can not perform efficiently. Also, there is no provision in serverless compute platforms to deploy and scale the same function instance with heterogeneous memory configurations; the allocated memory configuration remains constant across scaled instances.

No support for seamless deployment of the FaaS functions across the edge-cloud continuum: Current serverless compute platforms do not support seamless function deployments across the edge-cloud continuum. In fact, there are hardly any serverless solutions focussing on the edge-cloud continuum. The first documented efforts to bring serverless capabilities to the edge came from the industry with the introduction of AWS Lambda@Edge [297], allowing to deploy lambda functions to edge locations explicitly. This is then used within the IoT Greengrass system of Amazon [36]. It allows integrating edge devices with cloud resources in an IoT platform, and the application Lambda functions running on it are deployed to the edge computers. KubeEdge [314] is an open-source system extending native containerized application orchestration and device management to hosts at the edge. It only focuses on extending Kubernetes to edge devices for running containerized applications. There is no solution that allows the users to seamlessly deploy the FaaS functions across the edge-cloud continuum.

Absence of serverless computing over multi-cloud and hybrid-cloud: Serverless platforms do not allow the users to deploy or invoke the FaaS functions across multi-cloud or hybrid-cloud. In general, a serverless application consists of multiple heterogeneous functions and the resource (storage, memory, compute, and network) requirements for these functions are very dynamic and can differ vastly [185]. Serverless offerings from public cloud service providers have limitations. Such as in the AWS Lambda platform, a maximum of `1000` concurrent instances across all functions is allowed [186], while in Google Cloud Function (GCF) it is `3000` per-function [82]. Thus, a single cloud cannot meet the dynamically changing resource requirements of the functions. Thus, serverless computing over multi-cloud is necessary to overcome the limitations of a provider. Serverless computing over a hybrid-cloud can help with data protection and privacy laws and mitigate the effects of vendor lock-in [211]. Additionally, when compute demand exceeds the capacity of a private platform, cloud bursting to public platforms gives an organization additional flexibility to deal with peaks in IT demand.

Myriad of serverless compute platforms and their monitoring platforms: There exists a myriad of serverless compute platforms [260, 194, 116, 225, 217]. Each serverless compute platform is implemented differently and has a different virtualization stack. As a result, they perform differently [172]. Wang et al. [302] performed an in-depth study of resource management and performance isolation with three popular serverless computing providers: AWS Lambda, Azure Functions, and GCFs. Their analysis demonstrates a reasonable difference in performance between the platforms. Each platform has its monitoring solution; thus, different metrics and names, posing a challenge for monitoring across the edge-cloud continuum. One has to consider various monitoring data such as metrics data (related to the platform, application, and function level metrics) and traces of events based on OpenTracing standard. Some deployments are on-premise or at the edge; therefore, the infrastructure-based metrics must also be considered. Another challenge is bringing different metrics names into one standard form to compare with each other easily and to make the decision-making simplified. For collecting various metrics from the open-source platforms, one has to interface and extend the existing Kubernetes-based monitoring solution, such as Prometheus [295]. Furthermore, not all the serverless compute platforms can run on edge devices [233]. Therefore, one cannot run a homogeneous serverless compute platform over the continuum.

Serverless compute platforms do not account for the data access behavior of functions: Serverless compute platforms do not allow end-users much control over where a function is executed [273]. This becomes a problem when a function requires data not proximal to its execution location. It will cause data transfers and a significant idle period while the function waits for the data transfer to finish. Disregarding data locality when scheduling functions thus causes increased response times and inefficient network traffic, incurring more costs and potentially crippling Service-Level Objectives (SLOs). Moreover, with the growing market and popularity of the Internet-of-Things (IoT), many large applications are run on the edge-cloud continuum. The continuum combines computing resources in different locations and with varying constraints, making the control over function placement, and for that matter also data placement, a very desirable feature.

Absence of user-workload requests orchestration across multiple serverless compute platforms: There is no mechanism for orchestrating function invocations across multiple serverless compute platforms. Although, one can create a load balancer manually and use it to orchestrate the invocations across the platforms. However, not each serverless platform has the function endpoint in the same format. Furthermore, scheduling function invocations across the edge-cloud continuum require information on the function's invocations response times, and computation and data requirements. Developing a load balancing algorithm is another challenging task, since overall decision-making should not add extra overhead to the response time. Google Cloud Platform (GCP) has introduced load balancing of user requests to a serverless Network Endpoint Group (NEG) that consists of a Cloud Run, App Engine, or Cloud Functions service [236]. The load balancer serves as the frontend and proxies traffic to the specified serverless endpoint in this service. However,

serverless NEGs can point only to Cloud Functions residing in the same region where the NEG is created and is only restricted to their infrastructure.

No provision to optimally configure the memory of the FaaS functions within the application: When deploying a FaaS function on a serverless compute platform, users need to define memory configuration for their FaaS functions: a low-level information that directly influences the performance and cost of the serverless application [37, 276, 302]. Thus, the user has to make a trade-off analysis to define the suitable configuration for their required Service-Level Objectives (SLOs) [103]. Furthermore, there has been no provision to automatically configure the optimal memory of the FaaS functions within an application to adhere to certain SLOs [47, 98].

1.2 Dissertation Contributions

The aspects mentioned in §1.1 highlight some factors that make it difficult for users to adopt serverless computing for the edge-cloud continuum. *To this end, we propose [149] and develop an extension to the concept of Function-as-a-Service (FaaS) as a programming interface for serverless computing across the edge-cloud continuum. This extension is a network of distributed heterogeneous serverless compute clusters spread across the edge-cloud continuum called **Function Delivery Network (FDN)** analogous to Content Delivery Networks* [111]. A serverless compute cluster consists of a serverless compute platform on top of compute nodes deployed in a specific region, either at the edge, in the Cloud, or on-premise. FDN provides seamless integration across the edge-cloud continuum by allowing the user to deploy and invoke the functions across heterogeneous serverless compute clusters in the continuum. FDN also distributes the data required by the functions across the edge-cloud continuum by allowing the users to organize their data objects into storage buckets. Based on these, FDN provides **Function-Delivery-as-a-Service (FDaaS)**, which can deliver user workload functions invocations to a subset of serverless compute clusters spread across the continuum based on : 1) function-awareness, and 2) data-awareness. The invocations are then load balanced across the selected subset of clusters based on the set load balancing algorithm. The automatic management of resources in the proposed serverless-based FDN facilitates application development by shifting the burden to the cloud platform.

The main contribution of this work is the Function Delivery Network (FDN) presented in Chapter 4. Furthermore, we provide directions, tools, techniques, and practical experiences for FDN to function rightly. Figure 1.1 shows the schematic overview of the key contributions made in this dissertation with respect to the challenges listed in §1.1 and are briefly explained as follows:

1. **FDN supports serverless compute clusters creation and management over multi-cloud, hybrid-cloud and edge-cloud continuum**: We use Terraform [132] and Ansible [2] to automate the cluster's creation over multi-cloud, hybrid-cloud and edge-cloud continuum. Clusters created on public-cloud are based on Amazon Web Services (AWS) Lambda, and Google Cloud Function, while on private-cloud are based on OpenWhisk and OpenFaaS with OpenStack as the cloud infrastructure tool, and edge clusters are based on the OpenFaaS platform. For integrating and managing these clusters, we created a **FDN provider** in the virtual-kubelet [144], an open-source Kubernetes kubelet implementation that masquerades as a kubelet to create virtual Kubernetes worker nodes. These virtual worker nodes represent the serverless compute clusters in FDN, and the **FDN provider** then maps the pods created on virtual worker nodes to functions on underneath serverless compute clusters. By doing so, we leverage the Kubernetes capabilities to add, remove and manage multiple clusters within FDN. Furthermore, **FDN provider** contains the Application Programming Interfaces (APIs) for deleting and creating the functions on different serverless compute platforms. This allows to use a unified `kubectl-based` interface for seamless deployment of the FaaS functions across the continuum. (Chapter 4)

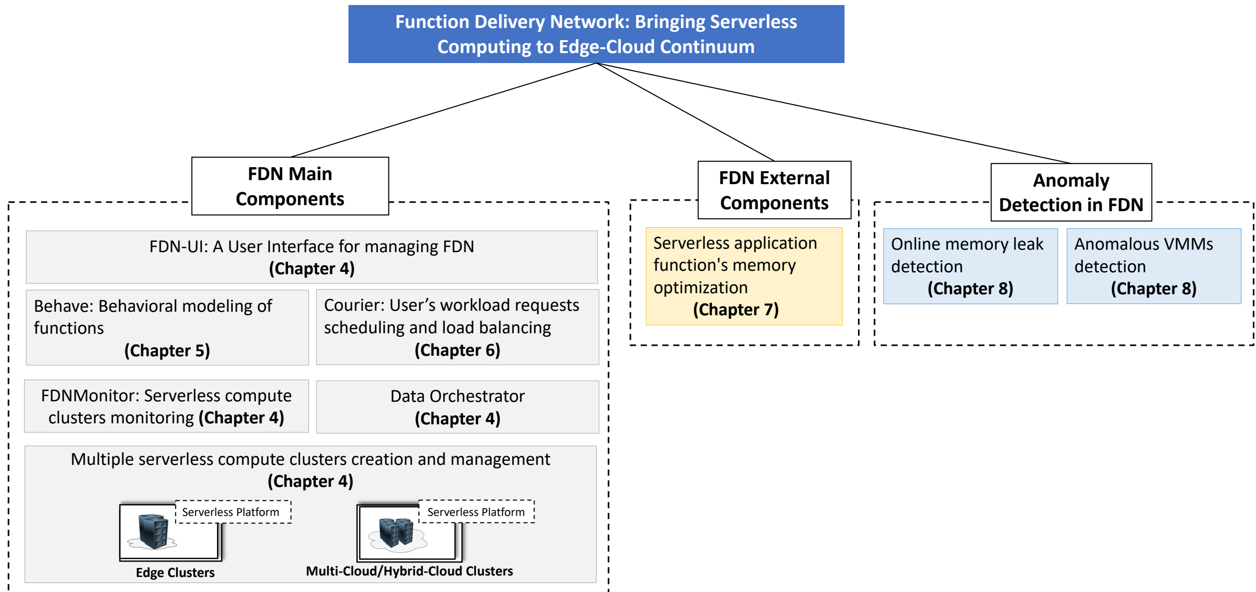


Figure 1.1.: A schematic overview of the contributions made in this dissertation.

2. **Multiple serverless compute clusters monitoring**: We create **FDN-Monitor**, a client-based python tool for monitoring various serverless compute clusters within FDN. It acts as a sidecar for every virtual Kubernetes worker node representing the serverless compute clusters in FDN. It collects various metrics classified into three categories: (i) User-Centric metrics, (ii) FaaS-Platform-Centric metrics, and (iii) Infrastructure-Centric metrics from the clusters. Furthermore, *FDN-Monitor* currently supports four serverless compute platforms: AWS Lambda, Google Cloud Function, OpenWhisk, and OpenFaaS. *FDN-Monitor* is designed modularly, and new serverless compute platforms can easily be integrated into it. (Chapter 4)
3. **Data Orchestration across the edge-cloud continuum**: We create the **Data Orchestrator** within FDN, which is responsible for managing the data across the storage backends in clusters within FDN. MinIO is selected as the object storage backend for each cluster in this work. MinIO is an AWS S3-compatible object storage technology that offers flexible bucket replication features [195, 196]. *Data Orchestrator* leverages MinIO's `mc` command line tool [200] and its JavaScript and Go language SDKs [199, 198] to integrate into FDN. It allows FDN distributing the data required by the functions across the continuum by allowing the users to organize their data objects into storage buckets. (Chapter 4)
4. **Behave: Behavioral Modeling of FaaS Functions in FDN**: We create two behavioral models based on the monitoring data collected by **FDN-Monitor** for characterization of the FaaS functions:
   - Functions Performance Model: For automatic end-to-end automatic performance modeling of functions, we develop a python-based tool called **FnCapacitor**. This tool automatically estimates the capacities of FaaS functions within a serverless application under the given SLOs and the memory configuration of the function instance. Furthermore, as part of **FnCapacitor**, we present a novel method which can be used to sandbox individual functions from the serverless application. Currently it supports Google Cloud Function (GCF), and AWS Lambda. (Chapter 5)
   - Functions Interaction Model: We model the serverless applications in the form of function compositions using neural Temporal Point Processes (TPPs). This is developed as part of a tool

called **TppFaaS** on top of OpenWhisk. **TppFaaS** uses two neural Temporal Point Processes (TPPs): 1) LogNormMix for providing the probability distribution of functions within the composition for the following function invocation, and 2) TruncNorm for predicting a function's invocation time. Such modeling and prediction can avoid cold starts by scaling functions in advance and reducing network load by optimizing the function-server assignment. (Chapter 5)

5. **Courier: Users's workload requests delivering and load balancing in FDN**: In order to distribute the incoming function invocations across the serverless compute clusters spread across the edge-cloud continuum, we create a system called **Courier**. **Courier** delivers the function invocations to the suitable subset of clusters in the continuum based on the function-awareness and data-awareness. Data-aware delivery takes the bucket name required by the function as the HTTP header parameter. **Courier** uses its value to select the appropriate subset of serverless compute clusters. The invocations are then load balanced across the selected subset of clusters based on the set load balancing algorithm. We have developed Latency-Aware, Service-Level Objective (SLO)-Aware algorithms and used it along with the default Round-Robin (RR) and Least Connections algorithms for load balancing. (Chapter 6)
6. **Serverless application function's memory optimization**: We build a tool called **SLAM** that automatically determines the optimal memory configurations for the FaaS functions within the given serverless application based on the specified SLO requirements. **SLAM** uses an optimization algorithm called SLAM-SLO along with its variants for various optimization objectives (minimum cost and minimum overall time) in addition to the SLO requirements in finding the optimal memory configuration for the given serverless application. **SLAM** currently supports AWS Lambda. (Chapter 7)
7. **Anomaly Detection in FDN**: Serverless compute clusters within FDN are created using Virtual Machines (VMs) hosted on the bare-metal server using a hypervisor. An anomaly in the application's functions deployed on those VMs can affect the availability and reliability of the application. Furthermore, a fault or an anomaly in the hypervisor hosting the VMs can propagate to the VMs hosted on it and ultimately affect the availability and reliability of the applications running on those VMs. Therefore, quickly identifying and eventually resolving it is crucial to save downtime. In order to do anomaly detection within FDN, we developed following two algorithms:
   - **Online memory leak detection**: We create an online machine learning-based algorithm called *Precog* to detect memory leaks on VMs. This algorithm only uses the VM's memory utilization as the primary metric. Our proposed algorithm achieves an accuracy score of 85% on the evaluated dataset provided by Huawei Munich Research Center and an accuracy score of above 90% on the synthetic data generated by us. (Chapter 8)
   - **Anomalous Virtual Machine Monitors (VMMs) detection**: In order to efficiently detect anomalous VMMs, we develop a machine learning-based algorithm called *IAD: Indirect Anomaly Detection*. It solely uses the resource utilization data of the VMs hosted on a VMM as the primary metric for the detection. We compare it against five other popular algorithms, which can also be applied to the described problem. It was found that the proposed IAD algorithm has an average F1 score of 83.7% averaged across four datasets and also outperforms other algorithms by an average F1 score of 11%. (Chapter 8)

1.3 Organization of the Dissertation

The rest of the dissertation is structured as follows.

Chapter 2 presents the background knowledge required for this dissertation. We start with the concept of virtualization, especially hardware-level virtualization (§2.1.1) and OS-level virtualization (§2.1.2). We

introduce the edge-to-cloud continuum (§2.2), discussing the edge and cloud deployment models. Since this dissertation is heavily centered around serverless computing, we also briefly overview the serverless computing model (§2.3) and the different serverless computing platforms (§2.3.2) used in this work. Lastly, we discuss the different cloud application architectures (§2.4).

Chapter 3 discusses the related work to this domain in seven folds. First, on the performance evaluation of microservices against the serverless applications. Second, the use of serverless computing for heterogeneous workloads in §3.2. Third, the performance variations among the serverless compute platforms in §3.3. Fourth, the use of serverless computing for the edge-cloud continuum in §3.4. In §3.5, we discuss the general cloud-based solutions for multi-cloud and hybrid-cloud along with specific works which exist in using serverless computing for multi-cloud and hybrid-cloud. In §3.6, we present the related work done in the field of data-aware scheduling in serverless computing. Lastly, in §3.7, we discuss the various solutions for optimizing the memory of serverless applications.

Chapter 4 introduces the Function Delivery Network (FDN) and its components. We start with FDN design overview, explaining the functional and non-functional requirements which FDN seeks to fulfil in §4.1.1, then the design methodology based on which we developed FDN in §4.1.2, and in §4.1.3, we present the final high-level overview of the FDN architecture. In §4.2, we explain each component of FDN in detail. **Chapter 5** introduces two behavioral models based on the monitoring data collected by *FDN-Monitor* for characterization of the FaaS functions: 1) Functions Performance Model (§5.1), and 2) Function Interaction Model (§5.2).

Chapter 6 presets Courier system, where we first explain the design of the *Courier Load Balancer* in §6.2. We present the *Courier Control Plane* in §6.3 along with the function delivery policies (in §6.3.1). In §6.3.2, we describe the latency-aware load balancing algorithm that balances the users' invocations across the subset of clusters selected based on the function delivery policy.

Chapter 7 presents the SLAM tool, used for finding the optimal memory configuration for a serverless application, consisting of several FaaS functions based on the specified SLOs. In §7.2, we describe it in details along with its components. We present the performance evaluation settings in §7.3.1 and evaluation results in §7.3.

Chapter 8 describes two anomaly detection algorithms: 1) Online memory leak detection using *Precog* in §8.1, and 2) Anomalous VMMs detection using *IAD: Indirect Anomaly Detection* in §8.2.

Chapter 9 explains the methodology used to carry out the performance evaluation. We first introduce the different benchmarks, i.e., FaaS functions, along with the developed application we use to evaluate in §9.1. Following this, we describe the different heterogeneous clusters used in this work to form the edge-cloud continuum within Function Delivery Network (FDN) in §9.2. Lastly, in §9.3, we describe the complete evaluation infrastructure and the different performance quality metrics used for the evaluation.

Chapter 10 presents the performance evaluation results of FDN. We first analyze the performance and resources usage variation of the FaaS functions on various clusters in §10.1. We summarize those results in §10.2 After this, we present the overhead introduced by the Courier in §10.3, FDN's function delivery policies correctness in §10.4, FDN's bucket replication performance in §10.5, and lastly FDN's load balancing performance in §10.6.

In **Chapter 11**, we finally conclude the dissertation and present an outlook on the future work.

2

Background

“The earlier you start working on something,
the earlier you will see results"

— Anonymous Author

In this chapter, we present the background knowledge required for this dissertation. We start with the concept of virtualization, especially hardware-level virtualization (§2.1.1) and OS-level virtualization (§2.1.2). We introduce the edge-to-cloud continuum (§2.2), discussing the edge and cloud deployment models. Since this dissertation is heavily centered around serverless computing, we also briefly overview the serverless computing model (§2.3) and the different serverless computing platforms (§2.3.2) used in this work. Lastly, we discuss the different cloud application architectures (§2.4).

2.1 Virtualization: a Cloud-Enabling Technology

Virtualization refers to the process of isolating a resource of a computer system and creating multiple “virtual versions” of the same [254]. Virtualization can be applied at various system levels: hardware-level, OS-level, and application-level [254]. Virtualization drives cloud computing economics [142] and since cloud computing is mainly based on the first two types of virtualization, we focus on hardware-level virtualization (§2.1.1) and OS-level virtualization (§2.1.2).

2.1.1 Hardware-level virtualization

In general, there are two types of hypervisors: 1) **Type I hypervisors**, and 2) **Type II hypervisors** [250]. Type I hypervisors run directly on the hardware to control and manage guest Operating Systems (OSs). It is also called a bare metal hypervisor, which runs natively on the hardware. A few examples of Type I hypervisors are Citrix/Xen Server [78], VMware ESXi [300], and Microsoft Hyper-V [137]. On the other hand, Type II hypervisors are usually installed on an existing OS. They rely on the host machine’s

OS to manage system calls and network, memory and storage resources. These are also named hosted hypervisors. Examples of Type II hypervisors include Microsoft Virtual PC [89], Oracle Virtual Box [4], VMware Workstation [301], and Oracle Virtual Machine (VM) Server [227]. Due to the host machine's OS presence in Type II hypervisors, a certain amount of latency is introduced compared to Type I hypervisors.

Cloud services providers mostly use Type I hypervisors on their physical servers to offer virtual resources over the Internet. For instance, Google Compute Engine (GCE) relies on Kernel-Based Virtual Machine (KVM) for virtualization [130], Microsoft customized Hyper-V called Azure hypervisor is used in their Azure cloud [133], and AWS uses their own hypervisor called AWS Nitro [43]. To understand how hypervisor partition and share the CPU, memory, and I/O devices to guest OSs, we discuss CPU virtualization (§2.1.1.1), memory virtualization (§2.1.1.2), and I/O virtualization (§2.1.1.3) in the following subsections.

2.1.1.1 CPU Virtualization

In the case of CPU virtualization, the idea is to virtualize the CPU resource of a computer system [180]. The sole purpose behind creating such a mechanism is to employ multiple smaller servers on a single large server. It helps to reduce the cost of hosting and, at the same time, enhances the utilization of the server. There are three different methods for implementing CPU virtualization:

Full Virtualization: The guest OSs are entirely abstracted from the underlying hardware by the hypervisor in full virtualization [180, 192]. Therefore, each VM and its guest OS operate as independent computers and require no modification. The hypervisor manages this by doing the binary translation of all the OS instructions at the runtime and caches the results for future use [75]. In contrast, user-level instructions run unmodified at native speed. Full virtualization offers the best isolation and security for virtual machines; however, the continuous translations between the physical and virtual resources, such as memory and processor, can impact the performance [274]. Microsoft Virtual Server is an example of it.

Paravirtualization: The guest OS kernel is modified to replace non-virtualizable instructions with hypercalls that communicate directly with the hypervisor in paravirtualization [180, 192]. In this scenario, the guest OS knows the virtual machine environment. It has a lower virtualization overhead and results in higher performance and efficiency than full virtualization. The open-source Xen project is an example of it [78].

Hardware Assisted Virtualization: It is an approach that enables efficient full virtualization using the help of hardware capabilities, primarily from the host processors. Intel Virtualization Technology (VT-x) [206] and AMD's AMD-V extensions [278] to the x86 architecture automatically trap the sensitive calls to the hypervisor, removing the need for either binary translation or paravirtualization. XenCenter [312], Linux KVM [158], and Microsoft Hyper-V [137] are some examples of hardware-assisted x86 virtualization. Linux KVM, an open-source Linux-based hypervisor, has a unique model. It is mainly classified as a Type I hypervisor. At the same time, the overall system is categorized as a Type II hypervisor due to a fully functional operating system. Thus, having the advantages of both Type I and Type II hypervisors.

2.1.1.2 Memory Virtualization

It is the process of sharing the physical host memory and dynamically allocating it to the VMs running the guest OS on it [180, 192]. It is done by virtualizing the Memory Management Unit (MMU), to decouple the physical host's memory into a pool of virtualized memory available to the VMs. The virtualized memory allocated to the VMs becomes their physical memory. The guest OS continues to control the mapping of virtual addresses in the guest OS to the guest memory's physical addresses. However, the guest OS does not directly access the actual host's physical memory. The Virtual Machine Monitor (VMM) is responsible for

mapping the guest's physical memory to the actual host's physical memory, and it uses shadow page tables to accelerate the mappings. VMM uses Translation Lookaside Buffer (TLB) hardware to map the virtual memory directly to the host memory to avoid the two levels of translation on every access.

2.1.1.3 I/O Virtualization

I/O virtualization allows a physical adapter such as a Network Interface Cards (NICs) or Host Bus Adapters (HBAs) to appear as multiple virtual Network Interface Cards (vNICs) and virtual Host Bus Adapters (vHBAs), respectively [73]. The ability to multiplex virtual I/O devices onto physical ones drives I/O devices to achieve better hardware usage. In general, there are three different ways to achieve I/O virtualization:

Emulation: Here, all the functions of a physical device, such as device identification, and Direct Memory Access (DMA), are emulated in software [73]. Emulation software is located within the VMM and appears as a virtual device in the guest OS. The I/O requests from the guest OSs are trapped in the VMM. VMM executes the I/O requests on behalf of the guest VM to the physical device and returns control to the VM. The computational overhead in emulating and trapping requests results in the host's moderate performance and high CPU utilization. Thus, emulation is mainly employed for simple peripherals, e.g., system timers.

Para-I/O Virtualization: The Xen hypervisor has popularized this approach. It is based on the cooperation between the guest OSs and the host. The guest loads a so-called frontend driver that communicates with a backend driver that the host operates. The process is also called a split-driver model. Payload data between the frontend and backend driver is transported via shared memory, and notifications about new data are exchanged via the VMM. Although para-I/O-virtualization achieves better device performance than device emulation, it has a higher CPU overhead. Additionally, para-I/O-virtualization can only be used if a frontend driver is available for the guest OS compatible with the host's backend driver.

Hardware-based I/O Virtualization: Hardware-based I/O virtualization can be achieved in multiple ways [101, 248]. Passthrough I/O, in contrast to emulation and paravirtualization, directly exposes physical I/O devices to VMs. It is done by mapping the memory regions of physical I/O devices to VMs. Enabling passthrough for untrusted VMs requires particular hardware extensions to retain spatial isolation requirements of virtualized systems. Therefore, modern x86 CPUs and chipsets come equipped with virtualization-enabled MMUs and Input–Output Memory Management Units (IOMMUs) (Intel VT-d, AMD). While passthrough I/O offers near-native performance, it has limitations in terms of scalability. Self-virtualizing I/O devices are an alternative to them [248]. The idea here is to offload penalties of software-based I/O virtualization and sharing routines into the I/O device hardware. This way, self-virtualizing devices can offer multiple virtual interfaces per physical device function, which are enabled by hardware-accelerated I/O virtualization. In line with these ideas, the Single Root I/O Virtualization and Sharing Specification (SR-IOV) was released by the Peripheral Component Interconnect (PCI) Special Interest Group (PCI-SIG) [92]. It is specified for Peripheral Component Interconnect Express (PCIe) topologies that utilize a single root complex. There is also Multiple Root I/O Virtualization and Sharing Specification (MR-IOV), a specification for topologies with multiple root complexes [283]. SR-IOV allows a PCIe device to appear as multiple separate physical PCIe devices. SR-IOV is based on the idea of Physical Functions (PFs) and Virtual Functions (VFs). PFs are full-featured PCIe functions; VFs are "lightweight" functions that lack configuration resources. VFs are attached to the guest VMs for carrying I/O requests, and they provide near native speed.

2.1.2 OS-level virtualization

It partitions the OS to create multiple isolated user-space instances [318]. A user-space instance is a virtual execution environment that can be forked instantly from the base operating environment. Such instances are

called **containers**. Programs running inside a container can only see the container's contents and devices assigned to the container. It is implemented using the standard Linux *namespaces* feature. In addition to isolation, the kernel often provides resource-management attributes to limit the impact of one container on other containers. The Linux *cgroups* provide resource management mechanisms.

Docker is an open-source container technology invented and developed by Docker Inc; that automates the deployment of applications inside containers [12]. Docker uses images as a base for containers; these images are similar to the VM images – they also contain software & OS that is already installed, configured, and tested [56]. Containers are instantiated from such images to run the software, e.g., a front-end server or a database. Docker containers can run on any host with a compatible OS as long as there is Docker installed and there are enough resources (CPU, memory). Docker containers became the de facto industry standard for containerized applications [56].

One needs a *container runtime* to run a container. A *container runtime* is a software that runs the containers and manages the container images on a deployment node. *containerd* is the default runtime used by the Docker engine [85]. It is often referred to as an industry standard because of its wide adoption. Underneath, this runtime uses `runc`, the reference implementation of the Open Containers Initiative (OCI) runtime specification. The OCI defines two standards – the *image-spec* for OCI images and the *runtime-spec* for system runtimes [102]. The typical job sequence of running a container would be that a container runtime (e.g., *containerd*) downloads an OCI image, unpacks it, and prepares a OCI bundle (a container specification including the root filesystem) on the local disk. After that, a system runtime like `runC` creates a running instance from this container specification. OCI images can be created using several tools, for example, `docker build` command. After the successful build, these images are usually pushed and published to a public or private container registry. *containerd*, runC [252], rkt [307], and lxc [308] are some examples of container runtimes.

Containers appeal to cloud users because of their lower booting time, portability, and direct resource management capabilities [205, 174]. However, they are often criticized for not providing strong isolation among containers on the same host. As a result, a new level of containers called VM-like containers is developed to hold the promise of strong isolation and minimal virtualization overhead [65]. They provide isolation capabilities similar to VMs; therefore, these are called VM-like containers. AWS Firecracker [5] and Google's gVisor [317] are some examples of it. The main difference between the two projects lies in the virtualization technology that forms the isolation layer [65]. Firecracker is a dedicated VMM implemented using Linux KVM. It emulates a minimal device to achieve low latencies when starting the VM and a low memory footprint on the host system. The VMs instantiated from it are often called MicroVMs. Firecracker does not start any containers in the MicroVMs; applications are directly run within them. Google's gVisor, runs containers using a new Linux Kernel, written in Go and running in the user space. This kernel intercepts the applications' system calls, thus providing additional protection from host kernel vulnerabilities. Figure 2.1 shows the overview of different isolation methods (VMs, Linux Containers, gVisor-based, and Firecracker-based) used to deploy the cloud applications. Firecracker and gVisor runtimes are used in the FaaS offering of AWS Lambda and GCF, respectively.

2.2 Edge-to-Cloud Continuum

With the emergence of cloud computing, the data is transferred to the cloud data centers through the network for computation and storage [62, 203]. However, with more and more IoT devices generating data, computation is being pushed towards edge devices forming edge computing [61]. On the one hand, the data is processed close to its source, decreasing the latency [91]. On the other hand, edge devices are usually

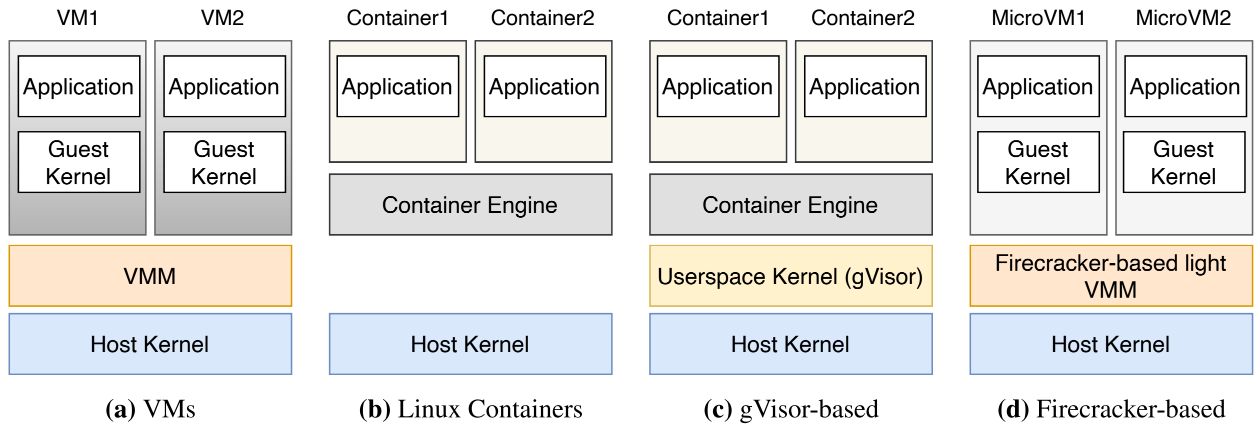


Figure 2.1.: Overview of four isolation methods (VMs, Linux Containers, gVisor-based, and Firecracker-based) for deploying the applications. VMs use a dedicated VMM such as Xen to provide isolation between them. Linux containers use the host kernel's namespace feature to provide isolation between the containers. gVisor-based containers are isolated using the userspace kernel. Firecracker-based MicroVMs use lightweight VMM based on KVM for the isolation.

limited in resources, limiting their compute power compared to the Cloud resources. The final architecture is a hybrid environment, connecting edge devices to the cloud and forming an edge-cloud continuum [48]. Computation offloading forMachine Learning (ML) applications, such as facial recognition algorithms, to the edge has shown considerable improvements in response times using the technology of split computing and early exits. Further research showed that using resource-rich machines called cloudlets near mobile users, which offer services typically found in the cloud, improves execution time [298, 255]. On the other hand, offloading every task may result in a slowdown due to transfer times between device and cloud so that an optimal configuration can be defined depending on the workload. Another use of the edge-to-cloud continuum is cloud gaming, where some aspects of a game run in the cloud while some are on mobile phones. Also, another potential use case is encrypting privacy-sensitive data at the edge before sending it to the cloud.

In this section, we introduce cloud (in §2.2.1) and edge computing (in §2.2.2) concepts in detail.

2.2.1 Cloud Computing

Cloud computing provides the on-demand delivery of IT resources such as compute, database, storage, network, etc., through the internet with pay-as-you-go pricing model [242]. These resources operate on servers located in large data centers worldwide. Cloud service providers like AWS, Azure, and Google Cloud Platform (GCP) are responsible for managing these servers. Traditionally, the IT team has to manage servers, apply patches to the operating system, develop and install the software, and ensure security. However, with the advent of cloud computing, cloud service providers offer numerous ways of using their IT resources, grouped into service models and cloud deployment models [114]. Each service model and deployment model provides different levels of abstraction, flexibility, and automation for varied tasks, thus providing more agility to the users. In the following subsections, we briefly describe various cloud deployment models (§2.2.1.1) and service models (§2.2.1.2) in cloud computing.

2.2.1.1 Cloud Deployment Models

There are four cloud deployment models, which represent the cloud environments :

Public Cloud: It is available to the public, and server infrastructure belongs to cloud service providers and is managed by them. Cloud service providers maintain and buy the hardware for the users. The IT resources are available as services, free of charge or on a pay-as-you-go pricing model via the Internet. Cloud users can scale-up or scale-down the resources on-demand. Public clouds include Amazon Elastic Compute Cloud, Microsoft Azure, Google App Engine, and IBM Cloud [171].

Private Cloud: In the Private Cloud, services are not available to the public but are intended for use solely by the owner company [242]. Resources are deployed on-premises using virtualization and resource management tools such as OpenStack. OpenStack is an open-source project that provides Infrastructure-as-a-Service (IaaS) capabilities for building a private cloud [258]. Several companies are building IaaS solutions on top of OpenStack [154]. This model, therefore, is sometimes called on-premises deployment. The infrastructure is maintained on a designated private network. This deployment model can provide dedicated resources and let users know where their data is kept and who has access to it. The major disadvantage of this model is the cost of purchasing and maintaining the hardware. Multiple public cloud service providers such as Amazon, IBM, Cisco, Dell, and Red Hat also provide private cloud solutions.

Hybrid: As the name suggests, it is a public and private cloud hybrid [309]. This model enables an organization to expand its infrastructure into the cloud while connecting to internal systems. For example, mission-critical workloads can run on a secure private cloud while less sensitive ones are deployed to a public cloud. This model encompasses the advantages of both public and private clouds. However, this model beneficial only if a company can split data into mission-critical and non-sensitive.

Multi-Cloud: It is similar to the hybrid cloud; however, multi-cloud uses multiple public clouds instead of merging private and public clouds [129]. In this model, users usually mix and match the best features of each cloud provider's services to suit their application and business demands. One of the disadvantages of this model is that the developers need to know the services of multiple cloud service providers, and the deployment strategy can get very complex.

2.2.1.2 Cloud Service Models

Each cloud service model provides a different level of control, flexibility, and management. Thus, choosing a suitable service model is an essential success factor for delivering cloud-based solutions. The following subsections focus on some widely adopted cloud service models [155].

Infrastructure-as-a-Service (IaaS): IaaS provides an abstraction over the tasks related to managing and maintaining a physical data center and infrastructure (servers, disk storage, and networking) [155]. It offers the services that represent the basic building blocks for cloud IT, such as networking, compute (virtual or on dedicated hardware), and storage as a collection of services that can be accessed and automated from code or web-based management consoles. IaaS comes with the highest flexibility and management control over IT resources. The user does not manage or control the underlying infrastructure, but has control over operating systems, storage, networking, and deployed applications. There are several IaaS vendors in the market, but the most widely used IaaS cloud service provider is AWS. An example of the IaaS compute service offered by the AWS is Elastic Compute Cloud (EC2), where a user can choose an instance type from various types that differ from each other by the amount of offered virtualized resources. For example, `t2.micro` EC2 instance type offers 1 vCPUs and 1 GB of memory in comparison to `t2.medium` instance type that comes with 2 vCPUs and 4 GB of memory.

Platform-as-a-Service (PaaS): Platform-as-a-Service (PaaS) provides the application platform consisting of infrastructure - servers, storage, and networking - and middleware, development tools, database management systems, and more [155]. PaaS avoids the cost and complexity of managing the underlying infrastructure and middleware. Users focus on the deployment and management of their applications. Despite having the advantages of PaaS, there are still some disadvantages. The users are provided with complete control of the platform; as a result, they still need to manage the scalability of their applications. Users have to give up a degree of flexibility because they are constrained by the tools and the software stacks that the providers offer. The users also have little-to-no control over lower-level software controls like memory allocation and stack configurations. Some examples of PaaS are: Google App Engine, container orchestration tools such as Kubernetes offered as Amazon Elastic Kubernetes Service (EKS) by AWS, Azure Kubernetes Service (AKS) by Microsoft Azure and Amazon Redshift data storage service by AWS. A new cloud deployment model called Serverless computing is also considered under PaaS. This dissertation uses it to a great length; therefore, it is described separately in §2.3.

Software-as-a-Service (SaaS): In this model, cloud service providers host end-user applications and make them available to the clients over the Internet [155, 86]. Cloud service providers are responsible for managing everything from the hardware to the software application. Cloud users are only responsible for using and bringing the data to the applications. Cloud service providers also ensure the application and data's availability and security. The users usually access these applications via a web browser. Two common examples of SaaS offers are web-based email and Customer Relationship Management (CRM) systems [86]. Some of the drawbacks of this model are data security and speed of delivery. Since data is stored on external servers, companies must ensure that it is safe and cannot be accessed by unauthorized parties. Slow internet connections can reduce performance, mainly if the cloud servers are accessed from far-off distances.

2.2.2 Edge Computing

To improve the response time and save network bandwidth, there is a tendency to push some of the computation and data storage closer to the data sources, along with the computation in the cloud [61]. This distributed computing paradigm at the network's edge is called edge computing [60, 256]. When looking at the edge computing landscape, there are myriad ways in which it can be used. However, these offerings can be boiled down to three distinct models described in the following subsections.

Managed by public cloud service providers: Cloud service providers such as AWS, Microsoft Azure, and Google provide regional extensions of their cloud platform to bring computational power closer to end-users. Existing customers hosting apps and data in one of these clouds can easily use those extensions. One such example of these offerings is AWS Local Zones [42], where local zones are available to users for application deployment. AWS Local Zones allow using selected AWS services, like compute and storage services, closer to end-users, providing low latency access to the applications.

Managed by LTE/5G telecommunications carriers: Telecommunications carriers are building out small-footprint data centers that can be leased to customers for edge computing purposes [55]. Additionally, telecommunications carriers believe that much edge computing will occur on Long- Term Evolution (LTE) or 5G mobile networks. Thus, the edge services deployed directly within the LTE/5G carrier's network provide the lowest-latency path. This is known as Multi-Access Edge Computing (MEC) and is offered by major telecommunications carriers like AT&T, T-Mobile, and Verizon [286]. Carriers are also partnering with Cloud Service Providers (CSPs) to combine the cloud provider's infrastructure architecture with the carrier's wider reach into the most popular metropolitan locations. For example, AT&T partnered with Google Cloud [19] and Microsoft Azure [20], while Vodafone and Verizon partnered with AWS to provide the service in their AWS wavelength offering [244].

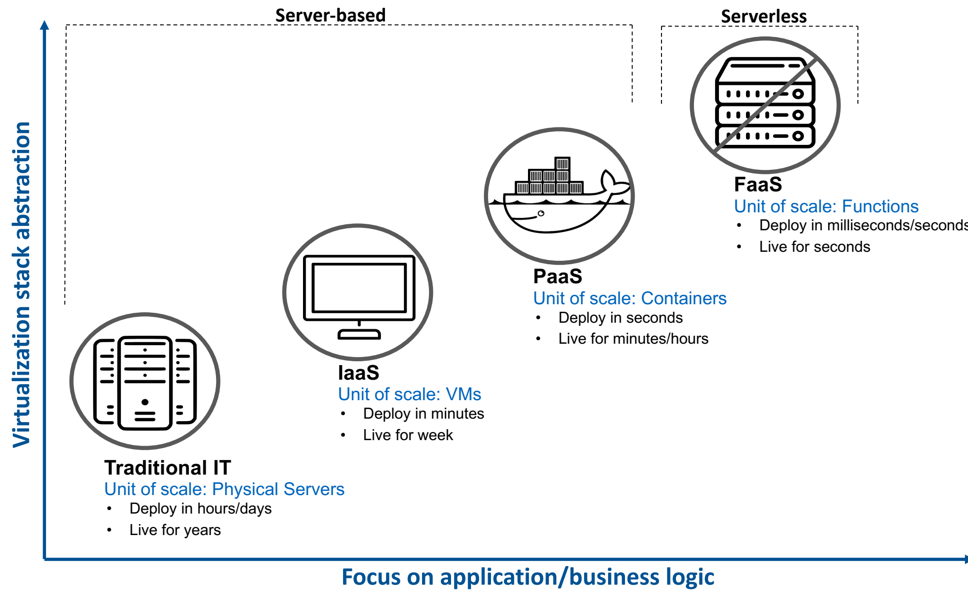


Figure 2.2.: Typical cloud service models comparison from the aspect of virtualization stack abstraction (y-axis) and focus on business logic (x-axis). Server-based here means that the user or application developer has to configure/manage certain infrastructure parameters. In contrast, the cloud service providers manage infrastructure entirely in serverless computing.

Hosted on-premises: Edge computing hosted on-premises comes under this category [159]. Cloud providers can also extend their public cloud within customers' private data centers. This is ideal for high bandwidth and low latency applications within the customer's private network. For Instance, AWS Outposts allows customers to deploy AWS-specific hardware into their private data center [106], while Google's Anthos [106, 15] allows customers to leverage their existing data center hardware and software. Microsoft offers both options with Azure private multi-access edge compute (previously private edge zones) [305] and Arc [303].

2.3 Serverless Computing

Serverless computing is a cloud computing model that abstracts server management and infrastructure decisions away from the users [304]. Significant progress has been made in different domains [72, 66, 265, 153, 109] based on the idea of *serverless computing* since its launch by Amazon as AWS Lambda in November 2014 [39]. In this model, the allocation of resources is managed by the CSPs rather than by *DevOps*, thereby benefiting them from various aspects such as no infrastructure management, automatic scalability, and faster deployments [80, 249]. Figure 2.2 shows the comparison of serverless computing model with different cloud service models from the aspect of virtualization stack abstraction (y-axis) and focus on business logic (x-axis). One of the biggest differences between other forms of cloud models and the serverless computing model is scalability [141]. In serverless computing, the application automatically scales up or down based on the resource usage (with scaling down to zero number of instances as well), and DevOps do not have to specify any scaling parameters. Furthermore, with serverless computing, the user does not have control of the platform, as is the case with other service models.

The Cloud Native Computing Foundation (CNCF) divides serverless into Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) [304]. FaaS is a key enabler of serverless computing [304]. FaaS provides an

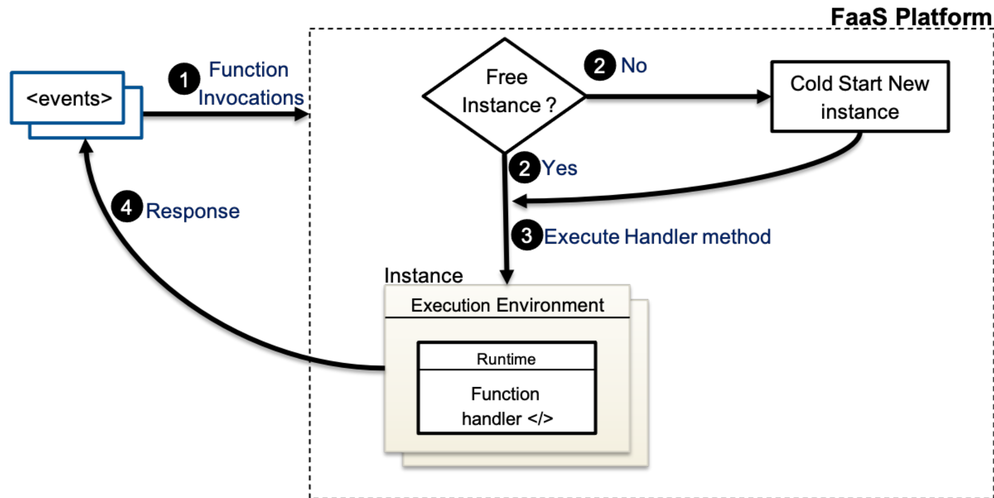


Figure 2.3.: Typical FaaS function invocation procedure. The first time the function is invoked, the serverless compute platform creates an *instance* of the function and runs its *handler method* in it to process the event. When the handler exits or returns a response, it stays active and becomes available to handle other events.

attractive cloud model since it facilitates application development in which the user does not have to worry about the infrastructure management but only about the code being deployed. The pricing is charged based on the number of requests to the functions and the duration, the time it takes for the function code to execute [41]. The latter varies according to the number of resources, such as memory and CPU cores allocated to the function, and are automatically adapted to deliver the best performance. Instead of developing application logic in the form of services and managing the required resources, the application developer implements fine-grained functions connected in an event-driven application and deploys them into the *serverless compute platform* such as AWS Lambda [37], GCF [79], and Azure Functions [11] for execution [304]. These functions are stateless, i.e., the state is not kept across function invocations. Functions can be invoked by a user's HTTP request or by another type of event created within the serverless compute platform. The platform is responsible for providing resources for function invocations and performs automatic scaling depending on the workload. The functions can be closely integrated with other services, e.g., cloud databases, authentication and authorization services, and messaging services. These services are called Backend-as-a-Service (BaaS) [304]. BaaS are the third-party services that replace a subset of functionality in a function and allow the users to only focus on the application logic [166]. In FaaS, function invocations are handled using containers or MicroVMs. Since functions are stateless, the application's state is stored in databases. The cloud service provider's infrastructure starts up on-demand ephemeral instances of each function. BaaS services are not set up to scale in this way unless the BaaS provider offers serverless computing, and the developers build this into their applications.

2.3.1 FaaS Function Invocation Procedure

FaaS-based function is a piece of code containing a *handler method* responsible for processing the *events* that are passed to the function when invoked, and these are executed within a *serverless compute platform*. FaaS-based functions can be invoked by a user's HTTP request or another type of event created within the serverless compute platform or the cloud infrastructure. These include changes to data in a database, files added to a storage system, or a new VM instance being created. The serverless compute platform is

responsible for providing resources for function invocations and performing automatic scaling. This is done by creating an *execution environment* which provides a secure and isolated runtime environment for the function. The functions can be written using various languages, and a language-specific environment called *runtime* is created in the execution environment. The runtime relays invocation events, context information, and responses between the serverless compute platform and the function.

The first time the function is invoked, the serverless compute platform creates a *Function Instance* of the function (execution environment) and runs its *handler method* in it to process the event. A *Function Instance* is an execution environment containing all the libraries and modules required for the *handler method* to execute. It is either based on the containers or MicroVMs. When the handler exits or returns a response, it stays active and becomes available to handle other events. If the function is invoked again while the first event is being processed, the serverless compute platform creates another *Function Instance*, and the two events are processed *concurrently*. As more events come in, the serverless compute platform routes them to available instances and creates new instances as needed. When the number of requests decreases, the serverless compute platform stops unused instances to have free scaling capacity for other functions. Serverless compute platforms usually have an upper limit on how many maximum concurrent instances called *Function Concurrency* can be created, such as 1000 for AWS Lambda (see §2.3.2.4) and 3000 for GCF (see §2.3.2.3). Figure 2.3 summarizes the overview of the typical FaaS function request procedure.

2.3.1.1 Cold-start Problem

FaaS-based functions suffer from the cold-start problem. It is mainly connected with loading the FaaS function into the executing server's main memory and preparing the target code's execution environment. The starting up of the VM/container and loading of libraries and function code constitute the cold-start latency [202, 63]. Several factors increase the cold-start latency of a function [188, 175]. One of these factors is the choice of the programming language. While languages such as JavaScript use an interpreter, Java requires a more complex JVM to be set up in the container, leading to higher latency. Another factor that has a decisive influence on the cold-start latency is the size of the function image. Suo et al. [281] propose HotC, a FaaS runtime management based on lightweight runtime containers. It leverages container runtime history and combines exponential smoothing and the Markov chain model to improve predictive accuracy. They maintain a live pool of already running containers for immediate reuse, since many functions would reuse the same base container image. These containers are cleaned up after the execution. Google Cloud Platform (GCP) provides a way to dynamically stream container images over the network [296, 1]. This allows the startup process of containers to begin before the entire image has been loaded. Furthermore, mature open source serverless compute platforms like OpenWhisk (see §2.3.2.1) use optimized caching and distinguish between cold, prewarm, and warm containers to address the cold-start problem [202]. Prewarm containers already have the runtime environment for the function. For example, when OpenWhisk's algorithm anticipates Node.js-based functions, it will start preparing generic Node.js containers, which reduces most of the cold-start time. When a function is executed very frequently, OpenWhisk will detect that and keep its containers warm. Warm containers are the containers where the function is already initialized and ready to be run at any time. Another approach for reducing cold starts is using dependency mining to precalculate the time when a function will be used in the future. This can be done by analyzing dependencies between different functions and/or by looking at historical invocations [279, 271]. AWS Lambda provides a way for pre-provisioning resources to the lambda function in order to save cold start time [187].

2.3.2 Serverless Compute Platforms

FaaS-based functions can be invoked by a user's HTTP request or by another type of event created within the serverless compute platform. The serverless compute platform is responsible for providing resources for function invocations and performing automatic scaling. Currently, many open source and commercial serverless compute platforms are available [204]. Serverless compute platform implementations are based on starting containers or MicroVMs for function invocations on top of a container orchestration platform such as Kubernetes. Applications are defined via a *deployment specification* that describes the functions, APIs, permissions, configurations, and events that make up a serverless application. The specification can be given via a command-line or web interface, or by using frameworks like Serverless [259] and Architect [16]. Updating of deployment is also done through this deployment specification. All the updates in the specification are instantly propagated, after which the containers are restarted, or only some configuration files are updated.

2.3.2.1 OpenWhisk

Apache OpenWhisk is a serverless open source cloud platform originally developed by a research group at IBM in 2015 and released in December 2016. It was later donated to the Apache Software Foundation [234]. It powers IBM's serverless offering, IBM Cloud Functions, and implements FaaS on top of Kubernetes as the container orchestration platform. Functions in OpenWhisk are called actions, and the execution of an action is called an invocation. Actions and rules can be created through the command-line interface (CLI) (`wsk` [14]), user interface (UI), or SDK. The actions can then be invoked either manually through the same methods or by event triggers. Events can originate from multiple sources, including timers, databases, message queues, or websites like Slack or GitHub.

OpenWhisk consists of multiple components under the hood, and all the components are packaged inside the docker containers when OpenWhisk is deployed [13]. Each function invocation is translated into an HTTP request to the Nginx server [208]. The Nginx server is a single point of entry, and its primary purpose is to implement the support for the HTTPS secure web protocol. On receiving a request, the Nginx server forwards it to the controller. The controller is responsible for authenticating and authorizing the requests in coordination with CouchDB, where all the user's data and privilege levels are stored. The controller also has a load balancer that keeps track of the availability of the invokers, i.e., the workers that run the code, and chooses one of them for the invocation. Controller and invokers communicate through Kafka [112], a publish-subscribe messaging system. The controller publishes the messages to Kafka addressed to a chosen invoker. Once the invoker confirms the message delivery, an HTTP request is sent back to the user with an *ActivationId*, which can be used for retrieving the results of this function call. This processing is asynchronous, and however synchronous processing is also available. It functions similarly to asynchronous processing, except in this case, the client will block until the action is completed and will retrieve the results immediately. Invokers set up a new docker container for each action, inject the code into them, execute the code, obtain the results, and then destroy it. These containers are run inside Kubernetes pods. There can be an invoker per Kubernetes worker node, or an invoker can be responsible for managing multiple Kubernetes worker nodes. Functions can also be chained into sequences, where chained functions use the output of the preceding function as input. OpenWhisk supports running functions in languages: Python, Node.js, Scala, Java, Go, Ruby, Swift, PHP, Ballerina, .NET, and Rust [226]. Functions not using these languages can be created by providing a custom-built docker runtime.

2.3.2.2 OpenFaaS

OpenFaaS is another widely popular open source serverless compute platform developed by OpenFaaS Ltd [217]. Until March 2019, it was developed by a team of full-time developers from VMWare [100]. It also implements FaaS on top of Kubernetes as the container orchestration platform. Functions in OpenFaaS can be written in any language, and unlike OpenWhisk, one does not have to create custom runtimes to make it work. A pre-built docker image of the function can be supplied to it. Similar to OpenWhisk, functions can be deployed manually or by setting up triggers through any interface to the OpenFaaS Gateway (CLI/UI/REST). OpenFaaS Gateway is the single point of entry for all the requests. From the gateway, CRUD (create, read, update, delete) operations and invocations are forwarded to the *faas-provider*, i.e., the controller, which translates OpenFaaS functionality to a certain provider. *faas-netes* [215] is an example of a *faas-provider* in OpenFaaS which enables Kubernetes for it. Because of this transparency to Kubernetes, one can interact with OpenFaaS resources directly through *kubectl*, the command line interface for Kubernetes. When a function is created, its code is pulled from the docker registry and executed inside a container. It utilizes *Prometheus* and its *AlertManager* to continuously expose metrics. The AlertManager uses these metrics to determine auto-scaling decisions and inform them to the OpenFaaS gateway, which then scales the function replicas up or down. The minimum (initial) and maximum replica count can be set by adding a label to the function at the time of deployment. When using Kubernetes, the built-in Horizontal Pod Autoscaler (HPA) can also be used instead of AlertManager [216]. Scaling down to zero replicas to recover idle resources is available in the OpenFaaS Pro version. This process is also called "idling" in OpenFaaS. The *faas-idler*, an external component, is responsible for making the scaling down to zero decision [214]. It monitors the built-in Prometheus metrics regularly along with the `inactivity_duration` variable to determine if a function should be scaled down to zero or not. Only functions with a label of `com.openfaas.scale.zero=true` are scaled to zero, all others are ignored. When using *faas-netes* as the provider, *faas-idler* is automatically deployed.

OpenFaaS's watchdog is responsible for starting and monitoring functions in OpenFaaS [218]. It provides a generic interface between the outside environment and the function. The watchdog is a tiny Golang web server that every function uses as their docker `ENTRYPOINT`. It acts as the initialization process for the function container. Once the function is invoked, the watchdog passes in the HTTP request via `stdin` and reads a HTTP response via `stdout` and sends it back to the user. OpenFaaS enables long-running tasks or function invocations to run in the background using Neural Autonomic Transport System (NATS) streaming [284]. This decouples the HTTP transaction between the caller and the function. The HTTP request is serialized to NATS streaming through the gateway as a "producer". The queue-worker acts as a subscriber and deserializes the HTTP request and uses it to invoke the function directly. To fetch the results from an asynchronous call, the user can specify a callback Uniform Resource Locator (URL).

2.3.2.3 Google Cloud Functions (GCFs)

Google Cloud Functions (GCFs) (know called Cloud Functions) is a serverless execution environment for building and connecting services in a cloud-based application offered by GCP [79]. With GCFs, developers do not need to provision any infrastructure or manage servers. The whole environment, including infrastructure, operating systems, and runtime environments, is managed by GCP. Currently, GCFs support JavaScript, Python 3, Go, and Java runtimes. GCFs are simple, single-purpose functions attached to events emitted from the cloud infrastructure and services. The function is triggered when an event being watched is executed. These events can be changes in a database, files added to a storage system, or the creation of a new VM instance. A response to an event is created using a trigger, which can then be attached to a function to capture and act on events. GCFs can be deployed using the web interface or the gcloud [113] command line

tool. Each GCF runs in its own isolated secure execution context, scales automatically, and has a lifecycle independent of other functions [119]. New incoming requests are assigned to function instances. Depending on the volume of requests and the number of existing function instances, a request may be assigned to an existing or a new instance. Each instance of a function handles only one concurrent request at a time. Thus, the original request can use the full amount of resources (CPU and memory) that is requested. In cases where inbound request volume exceeds the number of existing instances, multiple new instances are started to handle requests. This automatic scaling behavior allow GCFs handling many requests in parallel, each using a different function instance.

2.3.2.4 AWS Lambda

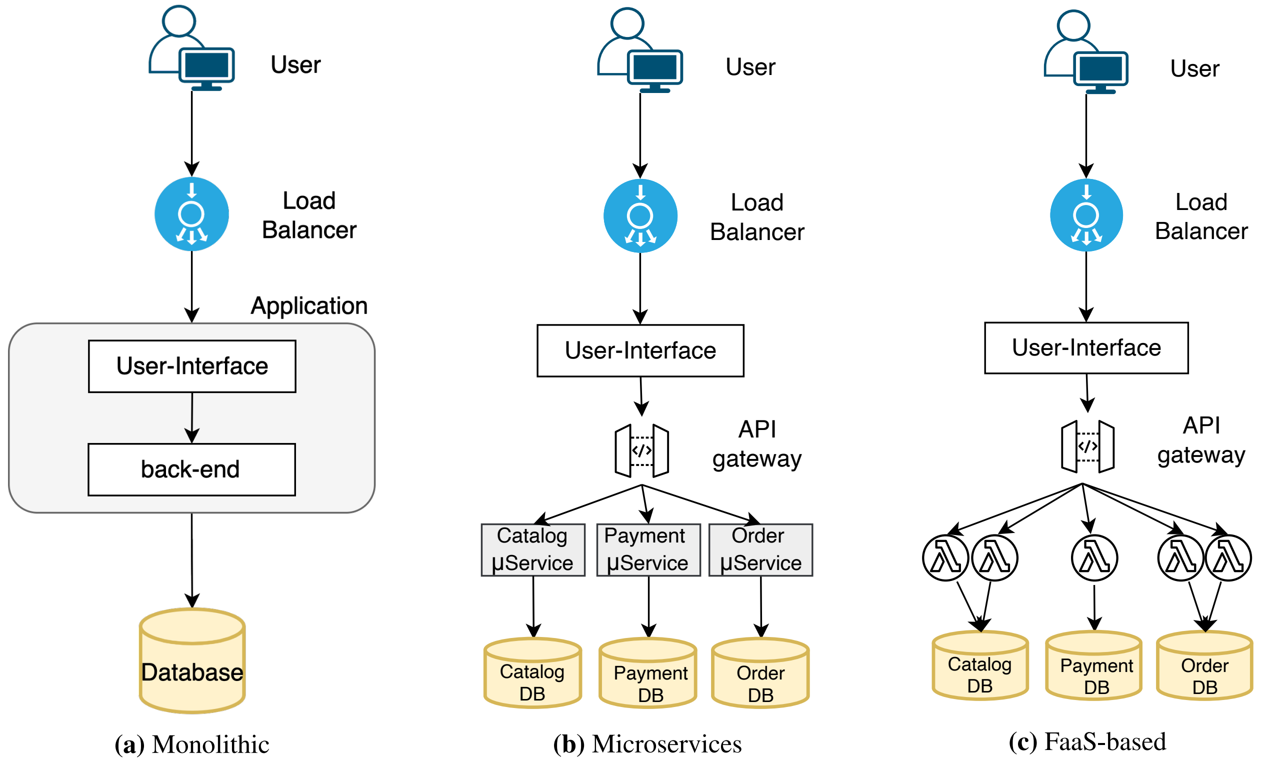
AWS Lambda is a high-scale serverless compute platform by AWS [37]. AWS Lambda functions can be triggered by various events on AWS like API Gateway. API Gateway can invoke a AWS Lambda function when it receives a HTTP(S) request. Another example of invoking an AWS Lambda function is when a new message is posted to a Simple Notification Service (SNS) topic. AWS Lambda also provides a dedicated HTTP(S) endpoint for the function called function URL, that can be used to directly invoke the function [165]. As events occur, the function code package is downloaded from the S3 bucket, installed in the runtime environment, and invoked. The runtime environment is based on an Amazon Linux Amazon Machine Image (AMI) [30]. The function code package contains at least the function code that will be executed when the function is invoked. However, it may also contain other assets that the code will reference upon execution, for example, additional files, classes, binaries, and libraries. When a Lambda function is invoked, code execution begins at the handler. The handler is a specific code method (Java, C#) or function (Node.js, Python) [28]. The handler can call other methods and functions within the files and classes uploaded as part of the package. It can also interact with other AWS services or make API requests to web services. There are two models for invoking a Lambda function: 1) Push Model - the function is invoked every time a particular event occurs within another AWS service, and 2) Pull Model – AWS Lambda polls a data source and invokes the function [30]. Also, an AWS Lambda function can be executed synchronously or asynchronously. AWS Lambda function also scales according to usage, but can be configured to throttle or increase concurrency if needed.

2.4 Cloud Application's Architectures

Traditionally, an enterprise application was designed in an n-tier architecture where the application is divided into "n" layers that perform logical functions, such as presentation, business logic, and data access. These layers are combined to form the monolithic application architecture. However, with the increasing demand for scalability and elasticity, an application is either decomposed into smaller services forming microservices application architecture or granular functions to form the FaaS-based application architecture. Since microservices and FaaS-based application architecture allow a design methodology that utilizes cloud services to build and run scalable applications in the cloud, these together form the cloud-native application architecture. In this section, we give a brief overview of each application architecture, starting with monolithic application architecture in §2.4.1, microservices application architecture in §2.4.2, and lastly, FaaS-based application architecture in §2.4.3.

2.4.1 Monolithic Application Architecture

Monolithic application architecture was one of the most widely used design patterns for enterprise applications [162]. Monolith means composed all in one piece. The application's components are packaged into a



(a) Monolithic **(b)** Microservices **(c)** FaaS-based

Figure 2.4.: Overview of three different application architectures.

single monolith (a massive structure) type architecture and deployed on the cloud in the monolithic application architecture. It is called multi-tier architecture because the applications are divided into three or more layers/tiers. For instance, Figure 2.4a shows an eCommerce application that authorizes a customer, takes an order, checks product inventory, authorizes payment, and ships ordered products. It consists of multiple modules like the front-end user interface and back-end modules like authorization, placing orders, inventory checking, and payment. All the modules are packaged as one application and deployed on the cloud. The whole monolith application can be scaled on-demand horizontally. The benefits of the monolithic architecture are: 1) easy deployment because the packaged application can be copied easily to multiple servers, 2) developers can efficiently conduct end-to-end testing on this type of architecture with automation tools, and 3) simple to scale horizontally by running multiple copies of the complete application behind a load balancer [162]. On the other hand, maintenance, reliability, technology refactoring, and scaling specific resources are some of the significant drawbacks of this architecture.

2.4.2 Microservices Application Architecture

In contrast to monolithic architecture, microservices architecture design is a more loosely-coupled style [54, 50]. The idea is to split the application into smaller, interconnected microservices. Each microservice is dedicated to a specific business goal and communicates with others through language and platform-agnostic APIs. These APIs are typically exposed as Representational State Transfer (REST) endpoints. Each microservice behaves as an independent, autonomous process without the dependency on other microservices. Each microservice may have its database or storage system, or they can share a common database or storage system. Microservices applications have the advantage that instead of launching multiple instances of the whole application, it is possible to scale-in or scale-out a specific microservice on demand. This allows

for providing a cost-efficient solution. Figure 2.4b shows the same eCommerce application discussed in §2.4.1 in microservices architecture design pattern. It shows that all the functionalities are isolated and decoupled into purpose-specific microservices with their databases. The front-end is separated from the core logic and can be scaled independently. Also, the back-end components are separated into different microservices with individual databases. Having individual databases for each microservices allows for modifying database schemas without impacting any other component in the infrastructure. Each microservice shown in Figure 2.4b can be placed behind its load balancers to achieve more throughput and availability of that microservice. The other benefits of a microservices architecture include improved fault tolerance and higher maintainability [212]. Although microservice architecture comes with many benefits, it also has some drawbacks. For instance, developers need to deal with the additional complexity of creating a distributed system by implementing an inter-process communication mechanism based on messaging or Remote Procedure Calls (RPCs). Also, deploying a microservices-based application is more complex since each microservice will have multiple instances, and each instance needs to be configured, deployed, scaled, and monitored. In addition, one needs to implement a service discovery mechanism as well [54, 50].

2.4.3 FaaS-based Application Architecture

Compared to the monolithic and microservices architecture, in FaaS-based application architecture, an application is decomposed into simple, standalone functions and uploaded to a serverless compute platform for execution. These functions are stateless, i.e., the state is not kept across function invocations. The function must contain a handler function, which may receive a payload upon which it can operate. Functions can be invoked by a user's HTTP request or by another type of event created within the serverless compute platform, such as triggers fired upon arbitrary changes in other cloud infrastructure components. Developers benefit from various aspects of this application architecture since it uses the serverless computing cloud model, which allows the benefits of no infrastructure management, automatic scalability, and faster deployments [105] as most of the infrastructure-based decisions are handled by cloud service providers. Figure 2.4c shows the same eCommerce application discussed above in FaaS-based architecture design pattern. The back-end microservices in the §2.4.2 are further decomposed into simple, standalone functions. Since there are no reserved instances in the serverless computing cloud model, the cost in this model is charged based on the number of requests received to the functions and the time it takes for the code to execute [38]. Thus providing the advantage of a lower total cost of ownership. In comparison to microservice applications, FaaS-based architecture has three advantages (1) no continuously running services are required, (2) functions are only charged when they are executed, and (3) the function abstraction increases the developer's productivity. On the other hand, some disadvantages of this application architecture pattern are cold-start problem [190], end-to-end testing, and vendor lock-in.

Generally, any application could be designed in monolithic, microservices, or FaaS-based architecture. Although all of them could deliver the application's functionality, scalability, security, interoperability, reliability, and maintainability of the application architecture are the significant differences among them. As a result, numerous aspects, such as the characteristics of the application, team size, and company culture, play essential roles when deciding on the application architecture [50].

3

Related Work

> *"Opportunities don't happen, you create them."*
>
> — Chris Grosser

This chapter discusses the related work to this domain in seven folds. First, on the performance evaluation of microservices against the serverless applications and discussing the architectural decisions on selecting microservices or serverless-based application architecture in §3.1. Second, since FDN targets heterogeneous workloads to be run on the clusters with heterogeneous serverless compute platforms, we present the prior work showcasing the use of serverless computing for heterogeneous workloads in §3.2. Then we present the prior work discussing the performance variations among the serverless compute platforms in §3.3. Fourth, FDN is designed as a serverless platform to work across the edge-cloud continuum; we present the prior work of using serverless computing for the edge-cloud continuum in §3.4. Fifth, FDN works across multi-cloud and hybrid-cloud; in §3.5 we discuss the general cloud-based solutions for multi-cloud and hybrid-cloud along with specific works which exist in using serverless computing for multi-cloud and hybrid-cloud. Sixth, in §3.6, we present the related work done in the field of data-aware scheduling in serverless computing. Lastly, in §3.7, we discuss the various solutions for optimizing the memory of serverless applications.

3.1 Microservices vs Serverless Applications

With the wide adoption of cloud computing, enterprises have migrated or refactored their existing monolithic-based applications into the microservices architecture [90]. This migration has affected the application's architecture and the team's structure within an organization [189]. Besides many advantages, microservices architecture also has some disadvantages in software development. For instance, each service communicates through the network via REST API endpoints, which can pose data security concerns during communication. On the other hand, serverless computing has gained higher popularity and more adoption in different fields [179, 98, 47, 153]. Compared to the monolithic or the microservices architecture, a serverless architecture releases the effort of server management from the application developers [70]. There are debates

about architecting decisions when choosing serverless or microservices application architecture [96, 285]. Jambunathan et al. [139] elaborated from the service deployment's perspective. They mentioned that the serverless-based deployment has infrastructure restrictions that need native cloud service support and must be hosted by cloud service providers. In contrast, a microservices-based deployment could be deployed on either the private data center or the public cloud. However, the benefits of auto-scaling without considering complex server configuration is a deployment advantage on serverless than microservices.

We conducted a thorough performance comparison of a cloud-native web application when deployed as microservices and FaaS-based functions from the aspects of scalability, reliability, cost, and latency [105, 146]. FaaS-based application in this work was deployed on AWS Lambda, GCF and OpenWhisk, while the microservices application is deployed on the Google Kubernetes Engine and AWS Fargate. We draw the following conclusions:

1. **Serverless strategy suffers from the cold-start problem**: When a function is triggered or invoked by a user request, the function is deployed in a newly-initiated instance. There is always a certain small period that a request must wait until the container is ready to serve. The instance usually takes this wait to initialize the environment and pull the function source code, called the *cold-start* problem. There already have been many kinds of research to decrease the cold start time like using pre-warmed containers [291], periodic warming consisting of submitting dummy requests periodically to induce a cloud service provider to keep containers warm [322] and pause containers [202]. DevOps need to consider this when deploying an application and decide whether this deployment strategy is beneficial or not based on the use case.
2. **Microservices deployment strategy suffers from the load balancing and traffic re-distribution problem**: Despite the cold start problem in the serverless deployment, it performed stably after the initial period. In contrast, microservices deployment had a high peak of duration scattered randomly during each test. One potential explanation is that these peaks coincide with scaling out or scaling in time of the autoscaling, which increases the response time. If one needs a stable latency over time, one could choose deployment using serverless computing.
3. **Microservices deployment strategy outperforms when fetching small size and repetitive requests**: For the API calls where the requests are with the simple payload and invoked repetitively having the static or small size response, they should leverage a microservices deployment. Serverless deployment has some minimum overhead due to the virtualization stack or the different involved components, which is more than what these cases need. As a result, for such cases, microservices deployment should be preferred.
4. **Serverless deployment is more agile in terms of scalability**: As we compare the scalability and agility of both deployments, serverless is better than microservices. Since the microservices deployment starts to auto-scale only after the system has reached the defined criteria for at least one minute, there is always a delay in responsiveness to re-balance the current workload. As a result, there is an increase in response time with the increasing workload, which drops after the new containers have been launched. In the end, the granularity of monitoring set at the minute-level limits the agility of the microservices scalability, which is not the case with serverless deployment. However, this disadvantage can be resolved by configuring a proper caching mechanism to store repetitive content, but the user must deal with more than required.

3.2 Heterogeneity in the FaaS Workloads

Serverless computing can be used for building a myriad of applications such as web applications, IoT, BigData workloads, chatbots, and Amazon Alexa, as well as IT automation [109, 72]. These serverless

applications generally consist of multiple heterogeneous functions [185]. Orchestration tools such as AWS Step Functions [261], Azure Durable Functions [193], or OpenWhisk's Composer [223] facilitate building such applications consisting of multiple functions. These functions can be arranged sequentially, in parallel, or in loops and integrate branching and conditional logic. Lynn et al. [183] studied seven different public serverless compute platforms including, AWS Lambda, GCF, and Microsoft Azure Functions, to showcase that serverless computing can be applied to a wide range of use cases. Splillner et al. [277] demonstrated that FaaS model could be used for different batch workloads, like, calculating the value of π, image face detection, password cracking, and weather forecasting. Serverless computing is highly relevant for scientific applications [153, 110]. Malla et al. [184] compared GCF with GCE in terms of cost and performance for a High-Performance Computing (HPC) workload. They found that FaaS can be 14% to 40% less expensive than IaaS for the same level of performance. However, the performance of FaaS exhibits higher variation due to on-demand CPUs allocation by the cloud service providers. There are some works using FaaS for data-intensive applications [311, 84]. Orfin et al. applied FaaS to latency-critical and user-facing applications and claimed to scale to millions of requests [228]. In our previous work, we also successfully used FaaS for achieving federated learning by using heterogeneous serverless compute platforms [72]. Based on these observations, we can say that FaaS workloads are heterogeneous, and the resource requirements for these functions are very dynamic and can differ vastly. Additionally, the resource requirements for the functions vary with changes in user input.

3.3 Heterogeneity among the Serverless Compute Platforms

FaaSProfiler [263] is the first to take a bottom-up approach in analyzing the architectural implication to unwrap the server-level overheads in serverless computing. They analyzed the difference between native and in-FaaS function execution and calculated the additional server-level overheads like computational overheads, memory consumption, bandwidth usage, and management overheads like orchestration, queuing, scheduling, and power consumed. Lee et al. [169] compared the performance of serverless compute platforms offered by public cloud providers by showcasing the results of throughput, network bandwidth, file I/O, and compute performance for the concurrent function invocations. L.Wang et al. [302] performed an in-depth study of resource management and performance isolation with three popular serverless compute platforms: AWS Lambda, Azure Functions, and GCF. Their analysis demonstrates a reasonable difference in performance between the platforms. They state that on Azure, 55% of the time, a function instance runs on a VM with debased performance. They also state that on Azure, the functions host VMs can have 1, 2, or 4 vCPUs. K. Figiela et al. [107] developed a FaaS function benchmarking framework where CPU-intensive functions were deployed on various serverless compute platforms. The authors observed fluctuations in response time for the identical deployments. Pawlik et al. [231] state that to assess the feasibility of running an application on the serverless compute platform, we have to determine the SLO of the application. It can be achieved by constructing a reliable performance model capable of analyzing a function performance, which requires knowledge about the performance of the infrastructure. Cloud service providers abstract details such as the number of cores, memory available, and network I/O capacity in the underlying hardware, usually limiting the available information to function time limit and maximum memory. The allocated memory also affects the provisioned CPU quota [179]. In our previous work [147], we developed a tool for estimating the maximum number of requests a microservice can handle when it is sandboxed. This capacity estimation of microservices enables us to ensure the flexibility of the capacity planning for a microservices application. These observations encouraged us to proceed with our work on estimating the capacities of functions when deployed with different deployment configurations (memory and maximum function instances) (§5.1). To our knowledge, none of the prior works take function_concurrency and sandboxing of FaaS functions into account when conducting research in the area of FaaS.

Furthermore, researchers have already identified the limitations of current serverless compute platforms, such as no control over specifying additional hardware resources like the required number of CPUs, GPUs, or other types of accelerators for the functions, and inefficient communication patterns between functions because of the data access latency [128, 47, 127]. Jonas et al. [152] suggest some improvements and workarounds which can be adopted to overcome these limitations. Since the FDN targets heterogeneous clusters, it can overcome these limitations by taking into account the cluster's heterogeneous resources (CPUs, and GPUs) and scheduling the function's invocations automatically to the right target cluster. In this process, one can also use FDN to replicate the data in the cluster where the function is scheduled. Shahrad et al. [263] studied the architectural implications of serverless computing and pointed out that the short function runtimes hamper exploitation of system architectural features like temporal locality and reuse in FaaS. We also examine the underlying processor architectures for GCF and determine the optimization of FaaS functions using Numba can improve performance by `18.2x` (geometric mean) and save costs by `76.8%` on average for the six functions [71]. PyWren [153] utilized an external ad-hoc orchestrator to share state and synchronize parallel execution of functions in simple map-reduce applications. There has also been some work to enhance the function startup latencies, such as SAND [7] in which the authors utilized application-level sandboxing and a hierarchical message bus for achieving shorter startup delays and efficient resource usage. McGrath et al. in [190] proposed a queuing scheme with workers, in which function containers that can be reused are put into warm queues, and workers where new containers need to be created are put into cold queues.

Pfandzelter et al. [233] highlight the problem of running cloud-based serverless compute platforms on edge and introduce a new platform called tinyFaaS for edge environments. In order to show the effectiveness of this new platform, they compared it to Lean OpenWhisk and Kubeless by deploying them on a Raspberry Pi. The main results show that the response latency and scalability are much worse in both cases than in tinyFaaS. Increasing the incoming load of requests on both platforms increments the error rate, while tinyFaaS could still satisfy all the requests. Another interesting serverless compute platform is BlastFunction, an Field-Programmable Gate Array (FPGA) sharing system designed for improving the performance of microservices and serverless application [87]. The main reason behind this project is to extend serverless computing to FPGAs to provide more hardware choices to the users. The performance of specific cloud applications may significantly increase with the utilization of FPGAs, since it would accelerate compute-intensive workloads.

These observations showcase the heterogeneity in the performance and resource availabilities of the various serverless compute platforms. Additionally, using the same platform for all types of clusters is not efficient, but it is better to develop specific platforms according to the needs. Thus, FDN across these heterogeneous serverless compute platforms can provide a way for enabling the scheduling of the functions on them by delivering the function invocations to the right platform based on its requirements.

3.4 Serverless Computing across the Edge-Cloud Continuum

In fog and edge [272, 229] computing, a considerable amount of research work has also been done to develop resource provisioning and management methods. Also, there have been studies on integrating edge and cloud computing for allowing the deployment of services on the resource-constrained edge devices and offloading compute-intensive parts to the cloud [299, 292, 124, 76]. Although the different proposed approaches for resource provisioning show promising results in traditional computing environments, they have not been evaluated and extended to be used with serverless computing. The first documented efforts to bring serverless capabilities to the edge came from the industry with the introduction of AWS Lambda@Edge [297], allowing one to explicitly deploy lambda functions to edge locations. This is then

used within the IoT Greengrass system of Amazon [36]. It allows integrating edge devices with cloud resources in a serverless compute platform, and the Lambda functions running on it are deployed to the edge computers. Satyanarayanan et al. [257] propose an edge computing approach to offload computation from mobile devices to the network edge using VMs based cloudlets. Bermbach et al. [53] have a very particular auction-based approach in which application developers bid on resource fog nodes to make a local decision about which functions to offload while maximizing revenue. It requires no centralized coordination and focuses on maximizing the earnings for the infrastructure provider. On the other hand, there is no guarantee for the user that its function will be executed. Baresi et al. [49] propose a serverless model for Multi-Access Edge Computing (MEC). They provide a broader range of application scenarios and optimizations, composing a serverless edge platform. KubeEdge [314] is an open-source system extending native containerized application orchestration and device management to hosts at the edge. These frameworks focus on executing the applications only on edge by extending cloud-based serverless compute platforms on edge. FDN includes *edge-clusters*, allowing the user invocations to be delivered to the functions deployed closer to the users, providing better performance. Furthermore, FDN allows multiple instances of the same function to coexist across multiple heterogeneous clusters, thus providing a way for handling function invocations from various opportunistic requirements.

3.5 Serverless Computing across Multi-Cloud

Although serverless computing is a new topic, it has been an active research topic [47, 127, 99, 123, 282, 53, 8]. However, not many have explored the direction of serverless computing across multi-cloud. In this section, we first present the general solutions connecting multiple cloud platforms in §3.5.1 and then solutions connecting multiple serverless compute platforms in §3.5.2.

3.5.1 Solutions Connecting Multiple Cloud Platforms

Brogi et al. present a software system called SeaClouds - Seamless adaptive multi-cloud management of service-based applications that tries to simplify the distribution, monitoring, and migration of PaaS software across multiple heterogeneous platforms [58]. The SeaClouds system deploys the different modules to the optimal platform, i.e., the platform that fulfills the requirements of the specific module. Additionally, it monitors the platform to ensure that it meets the requirements in the future. FDN consists of similar components, like SeaClouds. However, due to the nature of FaaS, we developed additional components like a load balancer. Apache Brooklyn is software to control applications' deployment, monitoring, and management in cloud environments [59]. It works by connecting different APIs and SDKs to provide a single software interface. It can conduct complex actions such as deploying a new web server instance and configuring the load balancer afterward. The developer can specify such actions via pre-defined policies and rules described in so-called *blueprints* in YAML syntax. Unfortunately, Apache Brooklyn lacks direct support for FaaS.

3.5.2 Solutions Connecting Multiple Serverless Compute Platforms

Aske et al. present a software system that helps developers define custom scheduling strategies [17]. To their service, they connected two public serverless compute platforms (AWS Lambda and IBM Bluemix) and one local OpenWhisk cluster. They implemented a low-latency scheduling algorithm that forwards requests to the cluster with the lowest Round Trip Time (RTT). Based on that algorithm, they can reduce

the overall computation time drastically. We use a similar approach in FDN to leverage the benefits of heterogeneous clusters by developing a latency-aware scheduling algorithm for orchestrating the invocations of the functions across the clusters. They do lack the support of seamless function deployment across the edge-cloud continuum, which FDN supports. Baarzi et al. introduce the concept of a virtual serverless provider (VSP) to allow customers to deploy serverless applications to different cloud providers through a consistent interface, hiding the differences from the users and helping them escape provider lock-in [46]. They also mention the issue of data locality and the importance of placing functions as close as possible to the data since this data can be costly to move or even illegal due to regulatory restrictions. Their primary focus is only on the function deployment, whereas FDN provides much more.

Furthermore, GCP has introduced load balancing of user requests to a serverless Network Endpoint Group (NEG) that consists of a Cloud Run, App Engine, or GCF service [236]. The load balancer serves as the front-end and proxies traffic to the specified serverless endpoint in this service. If the backend service contains multiple serverless NEGs, the load balancer balances traffic between these NEGs, thus minimizing request latency. However, serverless NEGs can point only to GCF residing in the same region where the NEG is created, and it is only restricted to their infrastructure. This is not the case with our implementation; *FDN* can work with heterogeneous clusters spread across multi-cloud and hybrid-cloud. Additionally, the load balancer to serverless NEG cannot detect if the underlying serverless resource (such as an App Engine, GCF, or Cloud Run (fully managed) service) is working as expected. This means that if a function deployed on GCF in one region is returning errors, but the overall infrastructure is operating normally, then the load balancer will not automatically direct traffic away to other regions. This is mitigated in our implementation by redirecting the traffic to other clusters in different regions, depending on their response times.

We first introduced the concept of FDN in 2020, consisting of a network of multiple heterogeneous target clusters orchestrated by a control plane capable of placing functions into several FaaS platforms [149, 150]. The FDN allows combining serverless compute platforms with different software and hardware characteristics. Doing so reduces overall energy consumption and provides better response times. To the best of our knowledge, there has been no implementation of FDN in literature, though it is urgent and a practical problem.

3.6 Data-Aware Scheduling in Serverless Computing

Serverless compute platforms do not afford end-users much control over where a function is executed. It becomes a problem when a function requires data not proximal to its execution location. It will cause data transfers and a significant idle period while the function waits for the data transfer to finish. Disregarding data locality when scheduling functions thus causes increased response times and inefficient network traffic, incurring more costs and potentially crippling SLOs. Latency-sensitive tasks, such as media streaming or complex distributed machine learning calculations, are thus not well suited to the current FaaS model [280]. A vision of functions as processes and the data center as a giant computer is presented by Al-Ali et al. [8] in a new abstraction called ServerlessOS. It aims to support not only event-driven computing but more general applications as well. Data management could be beneficial in this case, but the Edge nodes are not mentioned. Suresh et al. [282] present a function-level scheduler designed to minimize provider resource costs while meeting customer performance requirements. They do so by profiling the application and estimating the CPU shares. It is intended to be an option for existing baselines. They use a cloud-only solution, thus, not considering Edge nodes and data movement. Hellerstein et al. [127] describe FaaS as a data-shipping architecture because it still ships data to code rather than shipping code to data, and sees it as perhaps the biggest shortcoming of serverless compute platforms. The approach of fluid code and data placement, described as *stepping forward to the future*, is the suggested solution to the problem previously mentioned by

which the platform would physically colocate particular code and data. We designed the data replication in FDN based on this approach.

3.7 Memory Optimization of Serverless Applications

Many research works are aimed at optimizing the memory and cost for the FaaS functions. COSE [6] framework finds the optimal configurations for a FaaS function using the Bayesian Optimization algorithm while minimizing the total execution cost. It models not only the behavior of a function but also the environment (cloud, edge) in which those functions are deployed. However, they consider FaaS functions separately and optimize them based on cost. Bayesian Optimization was also used in CherryPick [9] tool for creating performance models for different cloud applications. The system provides 45-90% accuracy in finding optimal configurations and decreases costs up to 25%. However, they focused on traditional cloud applications. Another framework, Astra [143], is designed to optimize FaaS function configurations for specifically map-reduce usecase. Google and Amazon have also developed similar optimization tools. Google has developed a recommendation system to help the users choose the optimal VM type [118]. It currently does not support GCF. AWS Compute Optimizer [34] recommends optimal AWS resources for applications to reduce costs and improve performance by using machine learning to analyze historical utilization metrics. It can also be used to find optimal memory configurations for the lambda-based function. However, it can only be executed for the functions whose allocated memory level is less or equal to 1792MB and invoked at least 50 times in the last two weeks. AWS Lambda Power Tuning [69] tool uses an exhaustive search to identify the optimal memory level for a cost, or execution time. This algorithm will default need to perform at least 225 requests to the function to identify the optimal memory point. We also have developed a framework called MAFF [321], that uses numerous algorithms for various optimization objectives for automatically finding the optimal memory configurations of the FaaS functions.

None of the aforementioned research efforts address the issue of automatically configuring optimal memory of FaaS functions within a serverless application based on the user-defined SLOs. Most of the research addresses a single FaaS function or an application consisting of step functions that do not have complex call graph workflows. The proposed tool *SLAM* described in Chapter 7 fills that gap by creating a recommendation tool that in a short time can find optimal memory configurations of FaaS functions within a serverless application given the SLOs.

4

FDN: Function Delivery Network

> *"If you can't yet do great things, do small things in a great way."*
>
> — Napoleon Hill

We develop an extension to the concept of FaaS as a programming interface for serverless computing across the edge-cloud continuum. This extension is a network of distributed heterogeneous serverless compute clusters spread across the edge-cloud continuum called **Function Delivery Network (FDN)**. FDN provides seamless integration across the edge-cloud continuum by allowing the user to deploy and invoke the functions across heterogeneous serverless compute clusters in the continuum. FDN provides **Function-Delivery-as-a-Service (FDaaS)**, which can deliver user workload functions invocations to a subset of serverless compute clusters spread across the continuum based on : 1) function-awareness, and 2) data-awareness. The invocations are then load balanced across the selected subset of clusters based on the set load balancing algorithm.

In this chapter, we introduce the Function Delivery Network (FDN) and its components. We start with FDN design overview, explaining the functional and non-functional requirements which FDN seeks to fulfil in §4.1.1, then the design methodology based on which we developed FDN in §4.1.2, and in §4.1.3, we present the final high-level overview of the FDN architecture. In §4.2, we explain each component of FDN in detail.

4.1 FDN Design Overview

Before jumping into the implementation process and architecture of FDN, it was essential to carefully specify the core requirements to address the motivation goals (listed in §1.1) and which FDN needs to fulfill (§4.1.1). These requirements then led to an initial design proposal [149], which we improved over time following a design methodology described in §4.1.2. We present the final architecture of FDN in §4.1.3.

4.1.1 Requirements

In this section, we present the functional (§4.1.1.1) and non-functional (§4.1.1.2) requirements that FDN seeks to fulfill.

4.1.1.1 Functional Requirements

FDN seeks to fulfill the following functional requirements:

Multiple serverless compute platforms support: FDN should be able to support multiple serverless compute platforms from different public cloud providers and open-source ones: AWS lambda, GCF, OpenWhisk, and OpenFaaS. It is challenging since some platforms are based on Kubernetes, while some are leveraged directly from the public cloud providers. Each serverless compute platform has their own APIs and SDKs. Furthermore, due to resource constraints on edge devices, not all serverless compute platforms can run on them.

Management of clusters spread across the edge-cloud continuum: FDN should have the functionality to automatically manage the serverless compute clusters based on different serverless compute platforms spread across the edge-cloud continuum. Managing involves the addition, deletion, and update of the clusters. It should be designed in a scalable way so that the performance of FDN is not hampered, and new clusters could be added easily. Information related to the clusters that are part of FDN should be persisted over time. Additionally, FDN should be able to detect if clusters are available or not, including overloaded as well as offline clusters.

Seamless functions management across the clusters in the FDN: Each serverless compute platform has its own set of tools and APIs to perform function management operations. This makes it difficult for the users to create, delete, update and invoke functions on all the clusters based on the different platforms. Therefore, the FDN should provide the user with a standard interface for deploying, deleting, updating, and invoking the functions on all the clusters spread across the edge-cloud continuum. Furthermore, the user should be able to upload the code to the FDN, and the FDN should have the functionality to store it.

Data management across the clusters in the FDN: Data placement across the clusters must be considered to reflect the function's data affinity. Therefore, FDN should maintain information about the storage endpoints (currently, we only consider object storage, MinIO [201]) from all the clusters. FDN should be able to discover storage buckets across clusters and dynamically automate the replication of partial data sets across clusters according to user-specified configurations and changes in the data. This is specifically necessary for data-aware function delivery. FDN should provide a high-level endpoint to manage storage buckets and enforce user-defined constraints on data placement.

Clusters and functions monitoring: FDN should be able to gather performance metrics data on function invocations from all the clusters. It is especially relevant for the function delivery and load balancing across the subset of clusters described in Chapter 6. Furthermore, FDN should be able to monitor the platform, i.e., the serverless compute platform. For the clusters deployed either on-premise or at the edge, FDN should be able to monitor the infrastructure, i.e., collect infrastructure-based metrics. All the platforms have their naming conventions for each metric; therefore, FDN should be able to map the different metrics' names from each platform to one standard naming convention. It will simplify comparison and decision-making across clusters within FDN.

Function invocations delivery and load-balancing across the clusters in the FDN: The FDN should be able to receive function invocation requests and deliver them to a subset of serverless compute clusters spread

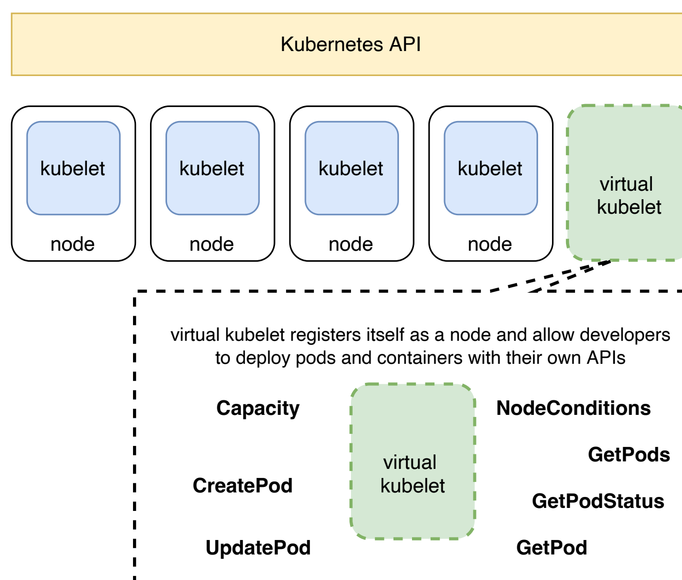


Figure 4.1.: A high-level design of *Virtual Kubelet*. *Virtual Kubelet* is an open-source Kubernetes kubelet implementation that masquerades as a kubelet to connect Kubernetes to other platforms [164].

across the continuum based on: 1) function-awareness and 2) data-awareness delivery policies (§6.3.1). Then it should be able to load balance the invocations across the selected subset of clusters based on the set load balancing algorithm (§6.3.2).

Behavioral modeling of functions in the FDN: The FDN should be able to support two behavioral models (described in Chapter 5) of functions: 1) performance model and 2) interaction model, deployed across the clusters using the monitoring data to enable autonomous orchestration decisions.

4.1.1.2 Non-Functional Requirements

Alongside the functional requirements defined above, FDN also looks to fulfill the following set of non-functional requirements:

1. **Scalability**: To handle more requests, FDN as a framework should be able to run across multiple nodes in a cluster without any modifications to the software.
2. **Extensibility**: The FDN should be easily extendable. It includes supporting new serverless compute platforms, adding new clusters, and implementing new function delivery and load balancing strategies.
3. **Performance**: FDN should perform efficiently, adding as little overhead as possible to function delivery and load balancing tasks.
4. **Minimize vendor lock-in**: FDN should provide an abstraction layer to vendor-specific APIs.

4.1.2 Design Methodology

One of the design decisions we had to make when implementing FDN was *how we could manage the multiple clusters spread across the edge-cloud continuum?* We initially developed a tool [145] using serverless framework [259] for managing serverless compute clusters with different platforms. A user could also use

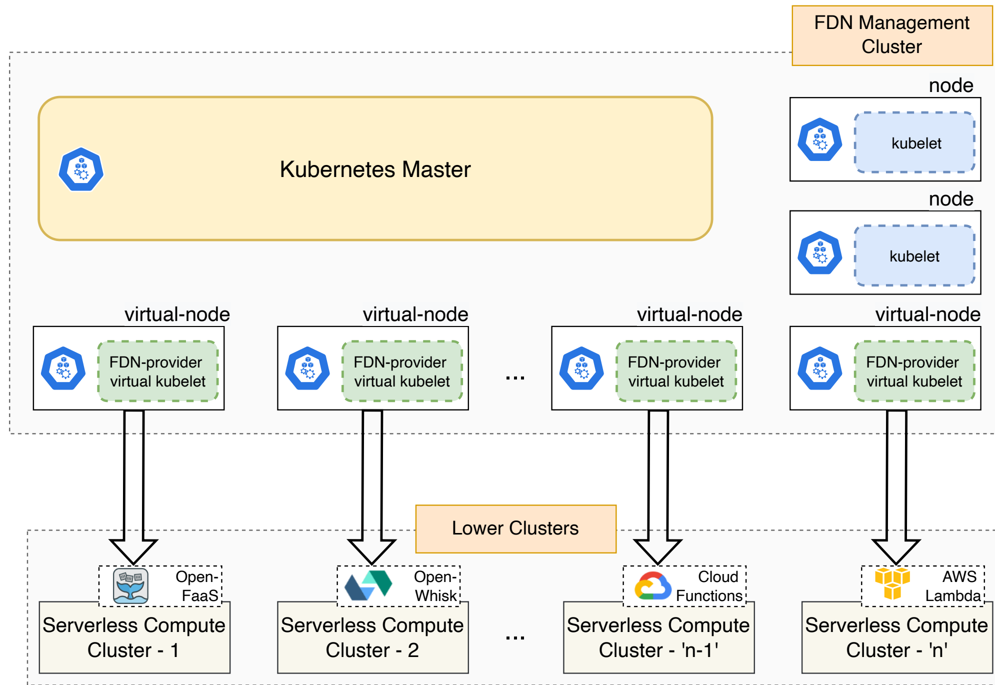


Figure 4.2.: A high-level design of *FDN-provider* in *Virtual Kubelet*. Every *Virtual Kubelet* node created using *FDN-provider* acts as a proxy for mapping to actual underneath serverless compute clusters. Pods created/deleted on virtual worker nodes are automatically mapped to functions in the underneath serverless compute clusters.

it for deploying functions to those platforms. However, it was not scalable, and the user had to specify many parameters. Then we thought that since Kubernetes is designed for managing containers and nodes across a highly distributed environment, why not leverage it to manage multiple serverless compute clusters? However, Kubernetes, by default, does not have the provision for managing clusters. Nevertheless, if we could replace a node within Kubernetes with a cluster and all the pods created on that node map as functions in the cluster, we would be able to manage the clusters. This design methodology leads our search to *Virtual Kubelet* [164].

Virtual Kubelet is an open-source Kubernetes kubelet implementation that masquerades as a kubelet to connect Kubernetes to other platforms [164]. It enables *Virtual Kubelet* to act as a proxy for Kubernetes to other platforms. Figure 4.1 shows the high level design of *Virtual Kubelet*. *Virtual Kubelet* registers itself as a virtual-node in Kubernetes and passes the API calls meant for those virtual-nodes to the corresponding platform. For *Virtual Kubelet* to function, one needs to write the translation of APIs such as pod management, pod status, and node status meant for virtual-nodes to the actual platform's APIs. These translations of APIs go in a pluggable `provider` interface within *Virtual Kubelet* that one needs to implement for defining the custom actions. The `provider` provides the back-end plumbing necessary to support the lifecycle management of pods, containers, and resources in Kubernetes [164]. There already exists several `providers` such as AWS Fargate Provider [23] for integrating with AWS Fargate service, Azure Container Instances Provider [45], and OpenStack Zun [219].

In order to integrate several serverless compute platforms, we created our custom provider called *FDN-provider* [144] for *Virtual Kubelet*. *FDN-provider* acts as federation for multiple serverless compute clusters. Every *Virtual Kubelet* node created using *FDN-provider* acts as a proxy to a serverless compute cluster. The serverless compute cluster could be based on any serverless platform like OpenWhisk, OpenFaaS, GCF, and

AWS Lambda. *FDN-provider* contains the APIs for managing the functions in different serverless compute platforms. It enables mapping the pods create or delete requests on virtual worker nodes to the functions create or delete requests on underneath serverless compute clusters. *FDN-provider* currently supports four serverless compute platforms: AWS Lambda, GCF, OpenWhisk, and OpenFaaS. Figure 4.2 shows the high level design of *FDN-provider* when deployed on a Kubernetes cluster. *FDN Management Cluster* represents the cluster where all components of FDN are running, and the lower clusters are the serverless compute clusters deployed with different platforms spread across the edge-cloud continuum. Such a design allows FDN to scale easily using Kubernetes features and incorporate new serverless clusters. For integrating a new cluster, one has to start a virtual-node customized for that cluster using a Kubernetes-based `deployment` configuration file with some command line parameters. The template of the `deployment` configuration file along with the command line parameters are shown in Listing A.1 in Appendix A. This template is automatically generated and applied when a new cluster is added and registered in FDN respectively. This design methodology enables using `kubectl-based` create and delete commands for creating and deleting functions on the respective serverless compute cluster. It is done by creating a Kubernetes `deployment` file for the function and applying it in the cluster. The Kubernetes `deployment` file template for creating a function is shown in Listing A.2 in Appendix A.

4.1.3 FDN High-level Architecture

In order to address the requirements described in §4.1.1, FDN has been designed following a highly modular and distributed architecture. It consists of several components and is divided into multiple layers. Figure 4.3 shows the overall architecture of FDN, with rows corresponding to different layers within FDN. FDN architecture is divided into six layers, described below (from bottom to top):

- **Infrastructure Clusters**: This layer corresponds to the serverless compute clusters part of FDN. These can either be created by FDN or created externally and then registered with FDN. The clusters are based on serverless platforms and are spread across the edge-cloud continuum. Each cluster is attached with a MinIO instance, where the data required by the functions are stored as objects in buckets.
- **Monitoring**: It is responsible for collecting monitoring data from the serverless compute clusters part of FDN and storing them into the database following a standard naming convention. It gathers function performance metrics, serverless compute platform metrics and infrastructure metrics.
- **Management**: This layer manages the functions, clusters, and data within FDN. The information related to clusters that are part of FDN, what functions are running on which clusters within FDN, and which storage buckets are located on which clusters are stored inside the FDN inventory database in this layer. The function's source code is stored inside the MinIO instance present in this layer.
- **Modeling and Scheduling**: Different behavioral models (Chapter 5) of functions deployed in FDN are build in this layer. Additionally, this layer is also responsible for delivering function invocation requests to the target clusters according to the specified function delivery policy (§6.3.1) and load balancing across the selected subset of clusters. Data corresponding to models and load balancer are saved in different databases within this layer.
- **Client**: This layer offers three perspectives to distinguish three different types of clients: user applications that invoke the FaaS function via user APIs, developers, and administrators/operators. The user client applications send function invocation requests to the load balancer, which handles requests' delivery and load balancing. Developers use the developer API to deploy, update or delete a function in the FDN. Administrators via the admin API manage the FDN administrative decisions.

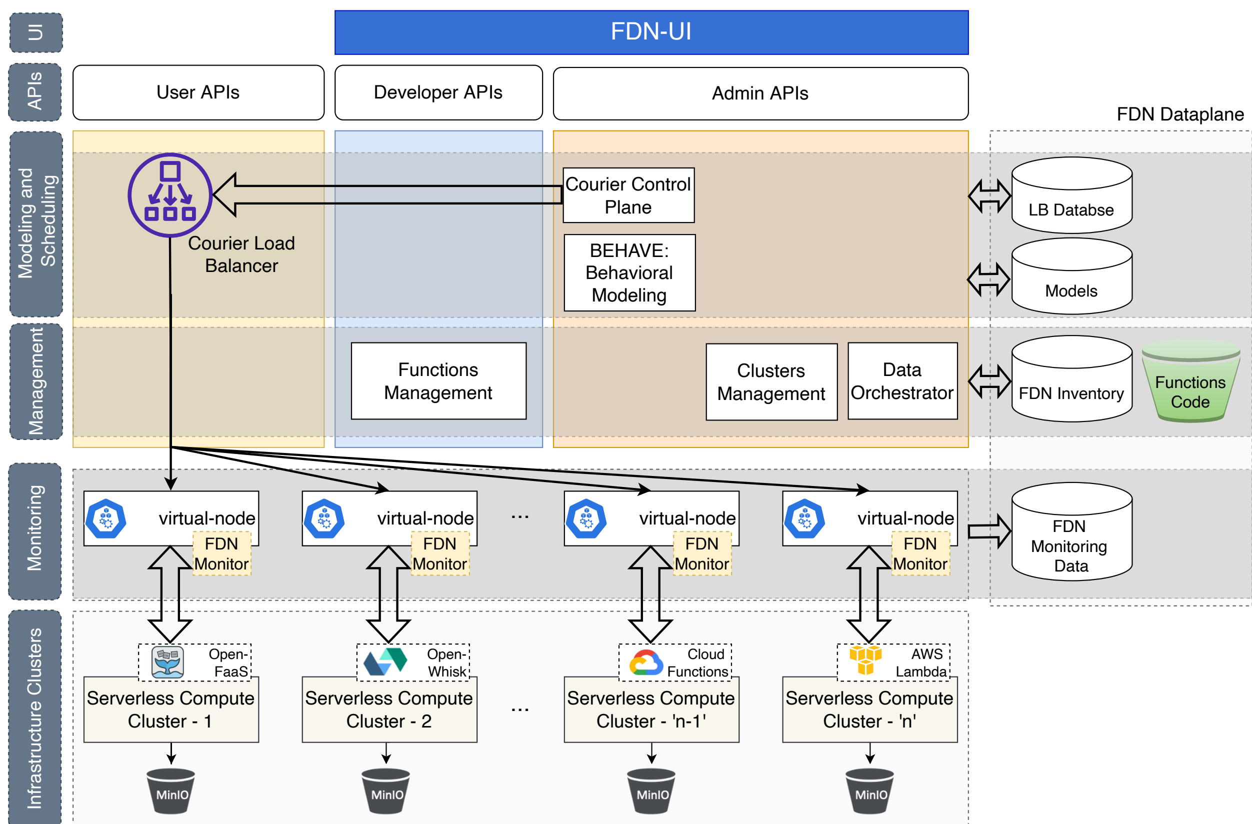


Figure 4.3.: A high-level architecture design of the Function Delivery Network (FDN). FDN architecture is divided into six layers, with each row in the figure corresponding to a different layer. FDN exposes three different types of APIs to distinguish three different types of clients: user applications that invoke the FaaS function via user APIs, developers, and administrators/operators. All the data within FDN corresponds within the FDN dataplane.

- **UI**: This layer corresponds to a frontend client to allow end-users to visually and intuitively interact with the FDN and affect changes through the FDN's APIs.

4.2 FDN Components

In this section, we describe each component of FDN in detail.

4.2.1 FDN's Serverless Compute Clusters

The serverless compute clusters are the operative component of the FDN and contain the deployed functions. A serverless compute cluster consists of a serverless compute platform (like AWS Lambda, GCF, OpenWhisk, and OpenFaaS) deployed on homogeneous compute nodes with specific hardware configurations either at the edge or in the cloud or on-premise [149]. The serverless compute platform is responsible for providing resources for function invocations and performing automatic scaling. Since FDN is designed to span across the edge-cloud continuum, the clusters within FDN are heterogeneous. For example, a cluster consisting of VMs in the cloud and another consisting of resource-constrained edge devices. Integrating

clusters with different computing power levels can potentially improve overall application performance. Different types of hardware may reduce the overall energy consumption by integrating IoT and other low-power target clusters [51]. In the same way, high-performance computing target clusters might add large amounts of computing power. Other domains where heterogeneous clusters can be relevant are edge and fog computing. Both domains include several types of hardware nodes, sometimes with a considerable difference in computing power (e.g., a smartphone and an AWS EC2 instance).

In order for a cluster to be incorporated within FDN, it must have the following:

- A cluster consisting of a serverless compute platform is deployed in the public cloud, on-premise, or at the edge. Four serverless platforms can be integrated into FDN: AWS Lambda, GCF, OpenWhisk, and OpenFaaS. These clusters should have an endpoint like a reverse proxy endpoint or a load balancer endpoint to which FDN can access. FDN currently does not influence the function execution within a cluster.
- Second, each cluster should have a monitoring solution deployed in them. We consider Prometheus [239] for on-premise and edge clusters, while AWS CloudWatch [81] for AWS Lambda and stackdriver monitoring [117] in GCF. This monitoring solution should expose an endpoint from which FDN can gather monitoring data periodically related to the cluster and platform.
- Third, each cluster is attached with a MinIO [201] instance, referred to as the storage endpoint. These storage instances store the data the functions require as objects in buckets. The storage access credentials are provided to FDN when the cluster is registered so that FDN can orchestrate the data. Currently, only MinIO based storage option is supported in FDN.

4.2.1.1 Cluster Types

We consider three types of serverless compute clusters based on the location they are deployed: at the edge or on-premise, or in the cloud. The following subsections describe each of them in more detail.

Edge Cluster: The *edge-cluster* consists of embedded devices with limited resources, such as Nvidia Jetson Nano [288]. Due to the limited resources available on these boards, it is not possible to run heavy serverless compute platforms like OpenWhisk. Therefore, for *edge-clusters*, we target OpenFaaS as it supports low-end devices and provides binaries for ARM processors. For running OpenFaaS, we need to run the Kubernetes cluster on the edge devices; therefore, we utilized k3s [243], a lightweight version of Kubernetes, to host a Kubernetes cluster on edge devices. k3s reduces the footprint and bootstrap-process of Kubernetes and combines all the low-level components required for running a Kubernetes cluster such as *containerd*, *runc*, and *kubectl* into a single binary.

On-Premise/Private Cluster: This cluster type is hosted on the VMs created on-premise using virtualization and resource management tools such as OpenStack. We first deploy a fully-fledged Kubernetes cluster on the VMs using *kubeadm* for this cluster type. Then we deploy an open-source serverless compute platform such as OpenWhisk or OpenFaaS in that cluster. This cluster type is essential for running functions that work on private data residing on-premise. Similarly, like *edge-clusters*, one needs to provision a master and multiple worker nodes here.

Public-cloud Cluster: We use a public serverless platform such as GCF or AWS Lambda for creating a *public-cloud-cluster*. These platforms do not provide internal configuration details of the VMs or the containers in which the functions are deployed. GCF, by default, provide an HTTP/HTTPs endpoint to the function. In AWS Lambda, previously, one needed to attach an API gateway to the function to get an HTTP/HTTPs endpoint. However, now by default, the AWS Lambda function gets an endpoint URL [68].

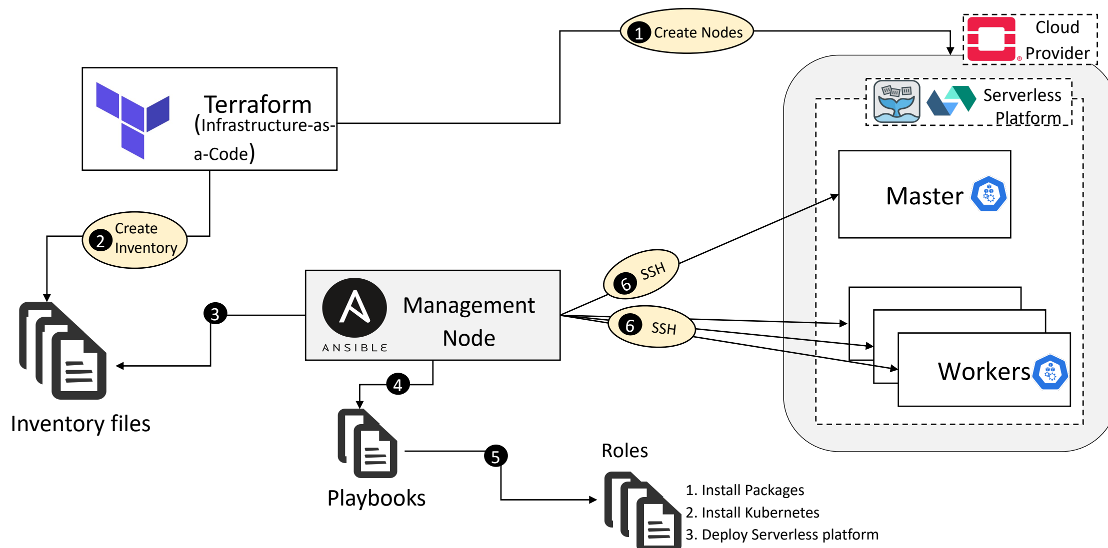


Figure 4.4.: Automation workflow for creating a serverless compute cluster hosted in a private cloud using Terraform and Ansible.

Furthermore, each of these platforms has a limit on the maximum number of function instances which can be created, `1000` for AWS lambda per account and `3000` for GCF per function. Thus, our clusters created on these platforms have these as upper limits.

4.2.1.2 Clusters Creation Automation

All the provisioning, deployment, and configuration of several clusters can be quite tedious and error-prone if performed by hand repeatedly. We, therefore, have developed automation to handle these tasks. These tasks differ for self-hosted clusters and public cloud clusters. All the edge and on-premise clusters require the deployment of Kubernetes and the serverless compute platform. Furthermore, for a private cloud-based cluster, one needs to provision and configure VMs as well. Therefore, we created the automation based on the popular tools Terraform [132] and Ansible [2]. Terraform enables infrastructure to be expressed as code in a simple, human-readable language called HashiCorp Configuration Language (HCL) [132]. As a result, the code can be versioned in a Version Control System (VCS) and track any changes made. Terraform is used for provisioning of VMs in the private-cloud cluster. Our private-cloud cluster resources are provided by Leibniz-Rechenzentrum (LRZ) [170] and it is based on OpenStack [258]. Therefore, we have used OpenStack modules within Terraform automation. Ansible, on the other hand, is an IT automation tool. It can configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates [2]. In this automation, Ansible is mainly used to create and configure a Kubernetes cluster and deploy a serverless platform in the cluster.

Figure 4.4 shows the workflow of the automation pipeline. It starts with Terraform provisioning the VMs on the cloud provider, grouped into master and worker nodes (step ❶). Based on these groups, terraform creates inventory files containing the VM's host information and SSH credentials (step ❷). Based on the group in these inventory files (step ❸), Ansible picks the playbooks (step ❹). Playbooks are the basis for a simple configuration management and multi-machine deployment system. They can also orchestrate steps of any manually ordered process and launch tasks synchronously or asynchronously [3]. Each playbook contains certain roles (step ❺). Roles are ways of automatically installing certain modules and running some tasks [251]. Ansible SSH into the VMs and runs the roles specific to the group the VM belongs to

(step ❻). In clusters where the servers or VMs are already provisioned, we start from step ❸. The link to the automation pipeline is specified in Appendix B in Table B.1.

We do not provision any VMs or install modules for serverless clusters based on the public cloud. They are managed entirely by cloud service providers. We only need the API access credentials for managing the functions; therefore, we have not created the automation for them.

4.2.2 FDN-Monitor

In order to collect monitoring data from various serverless compute clusters within FDN, the first challenge is to find what metrics to collect. Serverless compute clusters within FDN uses different serverless compute platforms, and each platform has its monitoring solution, thus diverse metrics. In the following paragraphs, we describe the monitoring stack used within each cluster based on different platforms.

AWS Lambda based Cluster: Functions deployed in this cluster are automatically monitored and collected by AWS CloudWatch [81]. Amazon CloudWatch is a monitoring service for resources, and the applications run on AWS. Therefore, we do not have to deploy any monitoring solution for it. We extract various metrics from it via REST API calls.

GCF based Cluster: Here also, the functions are automatically monitored by the cloud monitoring solution of Google Cloud called Google Cloud's operations suite (formerly Stackdriver [117]). Therefore, we do not have to deploy any monitoring solution for it as well. We again extract various metrics from it via REST API calls.

OpenFaaS and OpenWhisk based Cluster: Here the platform is deployed on top of Kubernetes; therefore, we have used a monitoring stack with Prometheus [239] for them. We have deployed two Prometheus instances in each cluster with various exporters. One Prometheus instance for the Kubernetes cluster, with cAdvisor [64] and node-exporter [240] as exporters, to collect metrics from the containers and nodes within the cluster, respectively. While the second Prometheus instance is deployed for collecting various metrics from the platform itself. Both the Prometheus instances can be merged, but they are kept separated in our deployment.

For collecting a wide variety of metrics from serverless clusters using different serverless platforms, we have built a client-based tool called *FDN-Monitor*. *FDN-Monitor* is responsible for gathering monitoring data metrics under the following three categories:

- **User-Centric metrics**: This category corresponds to metrics that a user of FDN can observe. The 90-percentile (P90) response time of requests, and the number of successful and failed invocations served per unit time, are calculated as part of this class of metrics.
- **Platform-Centric metrics**: Here the metrics corresponding to the serverless compute platform are collected. These are the number of function invocations resulting from the received requests, number of function instances, maximum number of concurrent instances allowed for processing events, number of cold starts, the execution time of the function (excluding the startup latency), and the memory allocated to each function instance. These metrics differ from platform to platform in name and also in number. For platforms hosted on Kubernetes, we can collect the function's resource consumption metrics such as CPU, memory, Disk I/O, and network usage.
- **Infrastructure-Centric metrics**: Here, the metrics from the host machines in the cluster are collected. Therefore, this category only exists for clusters hosted on edge or on-premise. The amount and usage over time of static resources, such as the number of cores, memory usage, Disk I/O, and network usage of individual nodes within a cluster, are collected under this category.

Table 4.1.: The summary of the monitoring metrics from *Platform-Centric* and *Infrastructure-Centric* categories for all the four serverless platforms considered in this work, along with the name used by the *FDN-Monitor*. For all these metrics, the data is collected per unit of time. For platforms hosted on Kubernetes, the functions run as pods; therefore, we can also collect the resources' consumption metrics of pods.

Metric Category	FDN-Monitor Name	AWS Lambda	GCF	OpenFaaS	OpenWhisk
Platform-Centric	success_invocations	invocations	invocations	invocations	activations
	replicas	concurrentExecutions	active_instances	replicas	replicas
	concurrency	concurrency	max_instances	max_pods	max_pods
	cold_starts	-	-	-	cold_starts
	init_time	-	-	-	action_initTime
	wait_time	-	-	-	action_waitTime
	average_execution_time	duration	execution_times	functions_seconds	action_duration
	pod-mem-limits	memory	memory	memory	action_memory
Platform-Centric (Functions Resources Usage)	pods-mem-sum-bytes	maxMemoryUsed	user_memory	pods-mem	pods-mem
	pods-cpu-sum	-	-	pods-cpu	pods-cpu
	pods-file-descp-sum	-	-	pods-file-descp	pods-file-descp
	pods-network-transmit-bytes†	-	-	pods-nw-transmit	pods-nw-transmit
	pods-network-receive-bytes†	-	-	pods-nw-receive	pods-nw-receive
	pods-fs-read-bytes†	-	-	pods-fs-read	pods-fs-read
	pods-fs-write-bytes†	-	-	pods-fs-write	pods-fs-write
Infrastructure-Centric	avg_cpu_system	-	-	avg_cpu_system	avg_cpu_system
	avg_cpu_user	-	-	avg_cpu_user	avg_cpu_user
	avg_cpu_iowait	-	-	avg_cpu_iowait	avg_cpu_iowait
	avg_cpu_idle	-	-	avg_cpu_idle	avg_cpu_idle
	avg_memory_usage	-	-	avg_memory_usage	avg_memory_usage
	network_bytes_transmitted†	-	-	network_transmitted	network_transmitted
	network_bytes_received†	-	-	network_received	network_received
	disk_writes_bytes†	-	-	disk_writes	disk_writes
	disk_read_bytes†	-	-	disk_read	disk_read
	disk_read_iops	-	-	disk_read_iops	disk_read_iops
	disk_write_iops	-	-	disk_write_iops	disk_write_iops

† In Bytes

Table 4.1 shows the summary of the monitoring metrics from *Platform-Centric* and *Infrastructure-Centric* categories from all the four serverless platforms considered in this work, along with the name used by the *FDN-Monitor*. For all these metrics, the data is collected per unit of time. For platforms, which are hosted on Kubernetes, the functions run as pods; therefore, we can also collect the resources' consumption metrics of pods and are referred to as *Platform-Centric (Functions Resources Usage)* in Table 4.1.

FDN-Monitor is developed in python and takes advantage of different python libraries and SDKs, such as boto3 [24] for AWS Lambda, and `monitoring_v3` [29] for GCF. These libraries and SDKs take advantage of the APIs provided by different serverless compute platforms and cloud providers to collect different types of metrics. It supports the following four serverless compute platforms: AWS Lambda, GCF, OpenFaaS, and OpenWhisk. *FDN-Monitor* is designed modularly so that each platform has the same interface for the collection. Figure 4.5 shows the UML diagram describing the structure of *FDN-Monitor* for various platforms. *FDN-Monitor* is connected with InfluxDB [135] for storing all the collected data, since InfluxDB is highly efficient for storing timeseries data and Grafana [121] for visualization of that data. The general flow of the data collection within *FDN-Monitor* starts with the user defining the environment variables and configurations. These variables and configurations include all information required by the modules, such as the credentials required to connect to InfluxDB, the credentials required to connect to the serverless compute platform, and the platform type upon which the cluster is based (e.g., AWS Lambda). Upon

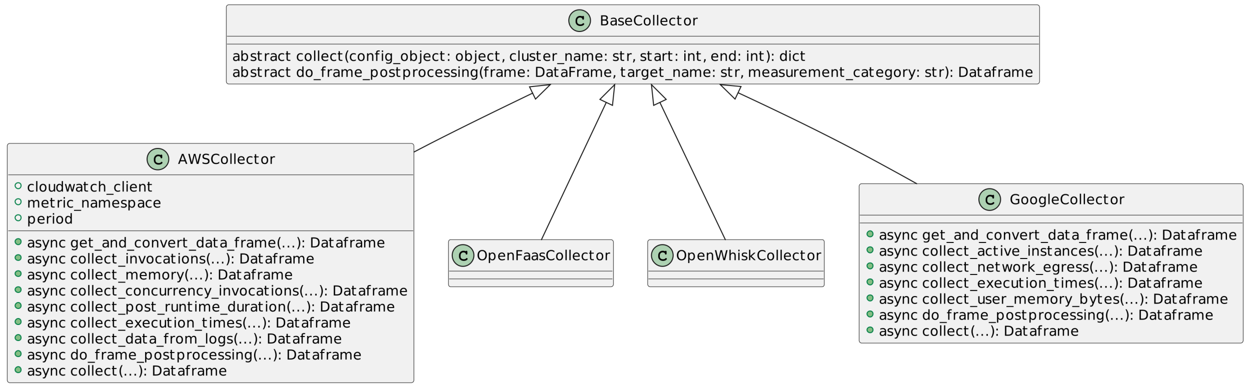


Figure 4.5.: A simplified UML diagram of *FDN-Monitor* showcasing the interfaces for data collection for the different platforms.

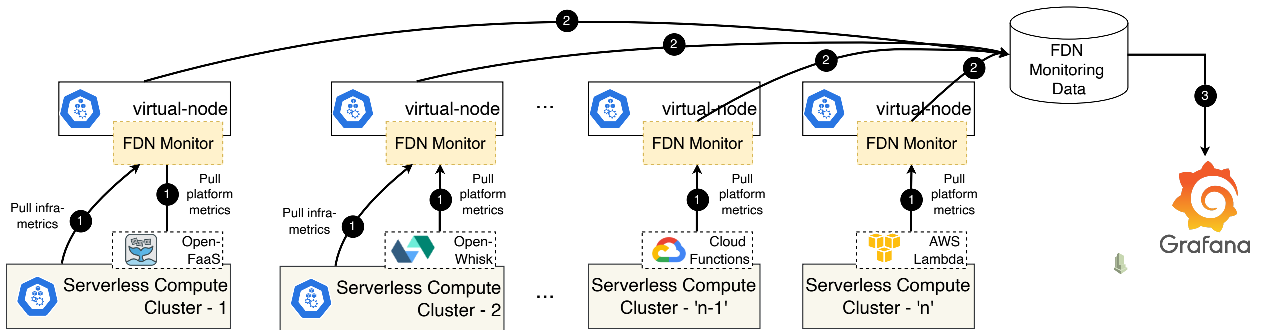


Figure 4.6.: Deployment of *FDN-Monitor* as a sidecar with every virtual-node in FDN. Each *FDN-Monitor* instance pulls the metrics from the underneath cluster and aggregate them into InfluxDB. Grafana queries the data from InfluxDB and showcase them in various dashboards.

reading the environment variables, the main function instantiates the correct type of data collector. For example, if the platform type is set to AWS Lambda, an instance of `AWSDataCollector` is instantiated with the correct AWS account credentials. In the case of AWS, the different `collect_<metric_name>(...)` (see Figure 4.5) functions are then called asynchronously. Similar interfaces exist for other platforms. Once the metrics are collected, they are combined into one pandas dataframe [241]. The collected metrics are filtered during the combination process to remove any empty results. For AWS Lambda based clusters, we have an additional function `collect_data_from_logs()`, which is responsible for collecting additional metrics provided by AWS logs insights - A feature for parsing logs provided by AWS. This allows for further platform-centric metric extraction, such as *billed duration* and *maximum memory used*. Responses from log insight queries are then processed and combined once again with the dataframe from the previous steps, based on the timestamp. For open-source based platforms, we have additional functions for collecting Kubernetes based metrics (e.g., `collect_pods_cpu_sum()`) and Infrastructure metrics (e.g., `collect_nodes_avg_cpu_usage_system()`). These functions are also called asynchronously, and results are combined with the dataframe from the previous step. Further, post-processing takes place, such as converting time columns to seconds, calculating mean values, and renaming columns to standardize the metric names. The dataframe is then stored in the InfluxDB and visualized using Grafana dashboards. We have created by default three Grafana dashboards, shown in Figure A.1 in Appendix A.

Figure 4.6 shows the deployment of *FDN-Monitor* in FDN.*FDN-Monitor* is deployed as a sidecar with every virtual-node (mapping to an actual serverless cluster) in FDN to collect the metrics from the clusters. After

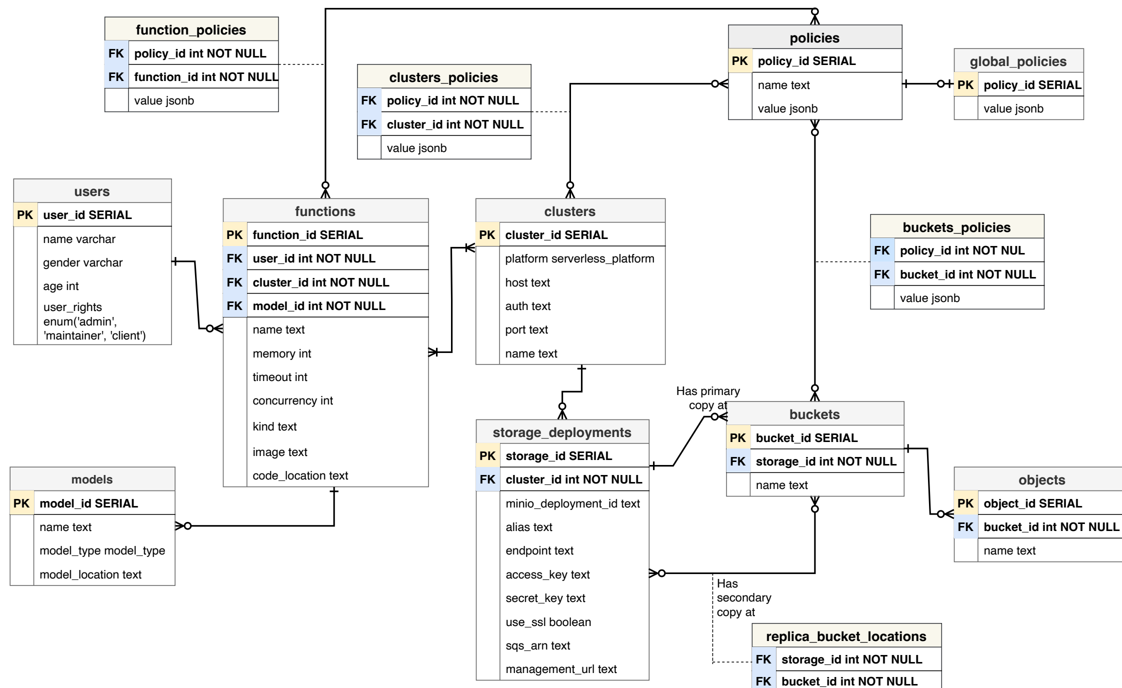


Figure 4.7.: FDN Inventory data model schema showing different entities as tables and relationships between them.

the deployment, each *FDN-Monitor* instance gets initialized by receiving the platform type and its credentials from the virtual-node. Based on the platform type, *FDN-Monitor* connects to the monitoring solution and initializes the data collection APIs. It then creates asynchronous tasks for pulling all the metrics from the underneath cluster's monitoring solution and waits until all the tasks are completed. The metrics are aggregated into a dataframe and stored in *FDN Monitoring Database*. *FDN Monitoring Database* is an InfluxDB database. The metrics collection procedure is periodically repeated every `30` seconds. During configuration, the collection period can be changed by specifying a different value for `DEFAULT_LOGGING_PERIOD` variable. Grafana queries the data from InfluxDB and showcases them in various dashboards.

4.2.3 FDN Inventory Database

The *FDN Inventory database* stores details concerning the different resources in the FDN: users, serverless compute clusters, MinIO storage deployments, the different storage buckets, and their contained data objects. It also stores information on the different bucket replication sets, detailing where the master buckets and replicas are located. Furthermore, the database contains information about different function models, function delivery policies, function load balancing algorithms, and user-defined bucket policies. Since FDN mostly contends with relational data, we selected PostgreSQL [289], a highly performant Database Management System (DBMS), as our database. Figure 4.7 shows the data model used by FDN. While some entities are easily recognizable as natural resources, such as a serverless compute cluster, others are more abstract, like a policy. The database schema uses tables and relationships to give concrete definitions to these entities and allows FDN to handle and manipulate them.

Policies: Policies are used in the data model to define specific settings. These can be global settings or specific to a cluster or a bucket. The schema defines the `policies` table, which lists the system's existing

settings, giving them a name and a default value, and the global_policies, clusters_policies, function_policies, and buckets_policies tables, which allow users to define policy values that should be applied globally, or to a given cluster, or function or bucket. This value overrides the default value set on the policies table. The value columns are all defined to contain a JSON object. It allows the application to define values flexibly, as the column can store different data types, such as a string, a number, or a more complex object.

Users: They are the actual users of the FDN and can be classified into different categories based on the APIs they use to interact with the FDN. users table stores the information about them and is also used for authentication and authorization when accessing FDN. Users are limited to a subset of the clusters defined by the admin user.

Clusters: Clusters are the records representing serverless compute clusters formed using a serverless compute platform, monitoring solution, and storage deployment. Cluster information such as type of the serverless compute platform, platform's URLs, etc. are stored in clusters table. Each cluster can be attached to many MinIO instances, and therefore we have a one-to-many relationship between the clusters and storage_deployments tables. Furthermore, the clusters_policies join table links clusters and policies together and forms a many-to-many relationship where a cluster can have many associated policies and vice-versa. It allows the definition of zones, which are conceptual delimitations defining the location of a cluster (edge, on-premise, or public cloud) useful for determining the clusters within a zone.

Storage Deployments: The storage_deployments table tracks all MinIO deployments across different serverless clusters within FDN. The table indicates the deployment's serverless cluster and the user-provided authentication data that FDN needs to communicate with the services and other relevant metadata it gathers. This additional metadata includes values like the minio_deployment_id, a unique ID that MinIO services generate for themselves and help FDN to determine the storage deployment.

Buckets: The buckets table lists all the storage buckets FDN is aware of amongst the different storage deployments across the clusters. These buckets are the containers for data objects and the grouping method used to organize data replication and scheduling. The buckets_policies table allows the definition of bucket-specific settings. For example, this is used to define a bucket's *allowed zones*, determining the zones to which a bucket can be replicated. Buckets only directly refer to the one storage deployment that keeps their master copy. However, the replica_bucket_locations table links the tables together to relate bucket replica locations. The schema allows FDN to determine a bucket's associated clusters through these means. As function invocations specify a storage bucket, FDN can choose where to forward the requests by listing the different clusters associated with the bucket.

Objects: The objects table lists the data objects present on the different storage deployments. Every object belongs to a singular bucket; their locations can be inferred by their bucket's master and replica locations.

Functions: The functions table stores all the information of the functions deployed in FDN across the clusters. This information includes name, memory required by the function, timeout, maximum function instances, function image path, and the MinIO object path storing the function code. Each function belongs to a user and can be deployed across multiple clusters in FDN. Furthermore, the function_policies join table links functions and function delivery policies together and forms a many-to-many relationship, where a function can have many associated policies and vice-versa. These policies dictate the clusters to which a function can be deployed, and invocations for it are delivered. Furthermore, multiple behavioral models are created for a function to model its different features.

Models: The models table tracks the different behavioral models created for a function and the locations of the models where they are stored.

4.2.4 Clusters Management

FDN's *Clusters Management* component written in Node.js is responsible for managing the clusters spread across the continuum within FDN. This management includes following methods:

Creation and update of clusters: *Clusters Management* allows the administrators using the FDN-UI or APIs to automatically create or update serverless compute clusters. This automation is based on the Terraform and Ansible explained in §4.2.1.2. Firstly, the administrator provides the required parameters as a YAML Ain't Markup Language (YAML) file or JavaScript Object Notation (JSON) data through *FDN-UI*, after which the input is validated. Then within the *Clusters Management* component, a container is initialized, having all the required dependencies for Terraform and Ansible. This container is responsible for creating or updating the cluster. Once the cluster is created or updated, it must be registered or re-registered as part of FDN. For that, *Clusters Management* requests internally a POST API endpoint `/api/clusters/register` with a JSON document shown in Listing 4.1. *Clusters Management* then saves this information in the *FDN Inventory Database*. Additionally, a Virtual Kubelet node is created or updated using *FDN-provider* customized for the cluster based on the template described in Listing A.1 in Appendix A. *FDN-Monitor* is also attached to it as sidecar (see Figure 4.6). Lastly, it notifies the *Courier Control Plane* to update the load balancer configuration. Figure 4.8 shows the workflow process of the cluster's creation.

Deletion of clusters: It is very similar to cluster create and update workflow. The administrator issues the request for a cluster delete through the *FDN-UI*. After which, the request is validated for having the required authorization. Then within the *Clusters Management* component, a container is initialized, having all the required dependencies for Terraform and Ansible. This container is then responsible for deleting the cluster. Once the cluster is deleted, it must be deregistered from the FDN. For that, *Clusters Management* requests internally a POST API endpoint `/api/clusters/delete` with the cluster name as body. *Clusters Management* then removes its information from the *FDN Inventory Database*, and it uses `kubectl delete` command to remove the Virtual Kubelet node. Lastly, it notifies the *Courier Control Plane* to update the load balancer configuration. Figure 4.8 shows the workflow process of the cluster's deletion.

Registration of clusters: This method is responsible for registering the clusters and saving their information in the *FDN Inventory database*. A POST API endpoint `/api/clusters/register` within *Clusters Management* is created for this purpose. It accepts the input as a JSON document shown in Listing 4.1. It then saves this information in the database and notifies the *Courier Control Plane* to update the load balancer configuration. During the registration, a Virtual Kubelet node is created using *FDN-provider* customized for the cluster based on the template described in Listing A.1 in Appendix A. This Virtual Kubelet node is a proxy for mapping to actual serverless compute clusters (see Figure 4.2). *FDN-Monitor* is also attached to it as sidecar (see Figure 4.6).

Listing 4.1: An example of JSON input recevied by the cluster register API endpoint for registering the cluster within FDN

```
1 {
2   "name": "name of cluster",
3   "platform": "serverless platformn name",
4   "host": "host endpoint",
5   "auth": "authentication",
6   "port": "port number"
7 }
```

Health Monitoring of clusters: Each Virtual Kubelet node representing the serverless compute cluster is attached with a health monitoring container. This container periodically requests an API endpoint in the

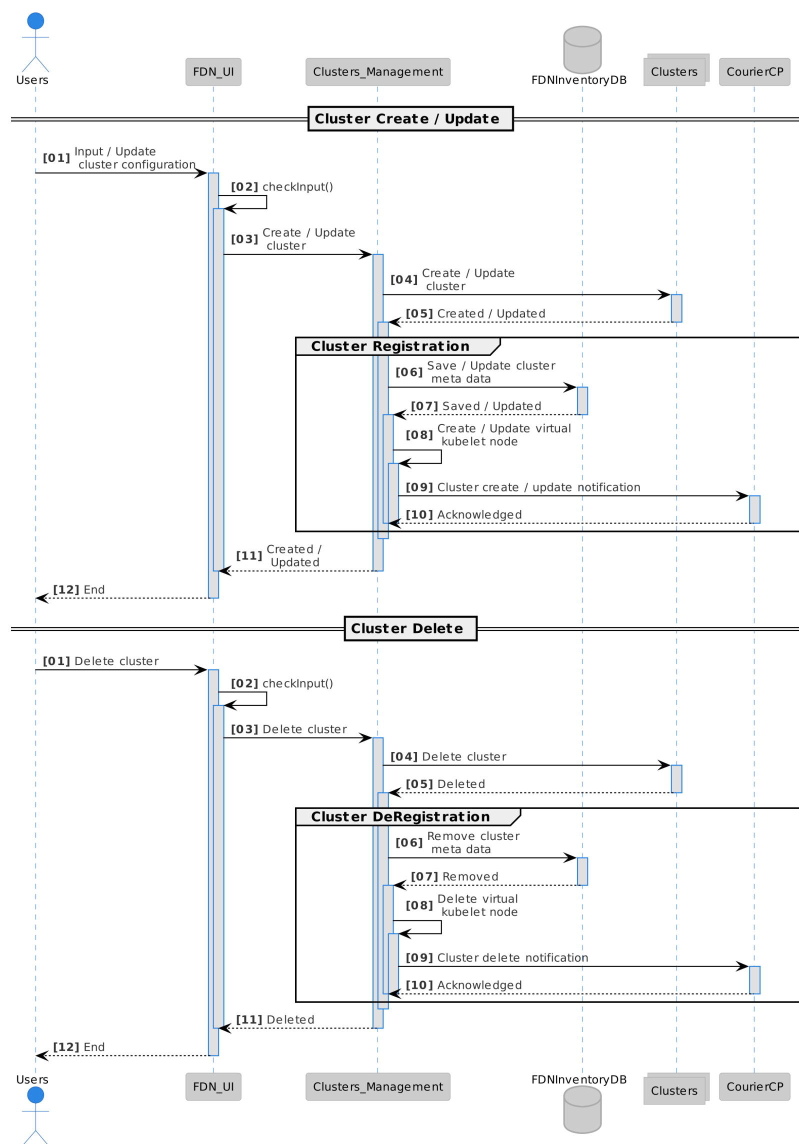


Figure 4.8.: FDN's Cluster Management workflow showing cluster create/update and delete.

cluster to check the health status. This API endpoint varies with the serverless compute platform. For serverless clusters based on AWS Lambda and GCF, we send the request to the function to get the status of the cluster. It may result in a cold start if the function instance is not warm. Based on the responses, the health status of the clusters is updated in the *FDN Monitoring Database*.

4.2.5 Data Orchestrator

Data Orchestrator within FDN is responsible for managing the data across the storage services in clusters within FDN. MinIO is selected as the object storage service for each cluster in this work. MinIO is an S3-compatible object storage technology that offers flexible bucket replication features [195, 196]. Its compatibility with Amazon's widely popular S3 storage solution makes it representative of current cloud storage technologies. Its bucket replication features provide building blocks to implement FDN's granular

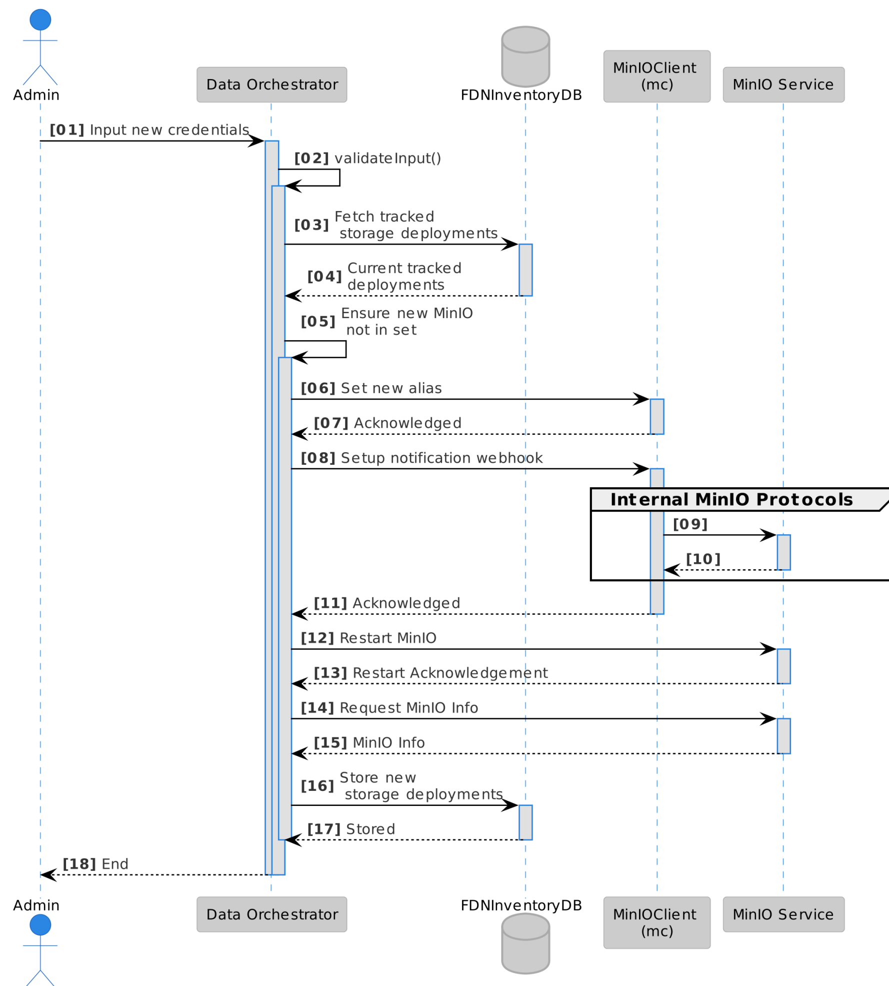


Figure 4.9.: Sequence of events to track a new MinIO deployment.

bucket replication mechanisms. *Data Orchestrator* leverages MinIO's `mc` command line tool [200] and its JavaScript and Go language SDKs [199, 198] to integrate together the different MinIO instances within FDN. With these tools and methods in place, the *Data Orchestrator* accomplishes four different operations:

Tracking a new storage deployment: To track a new MinIO service in FDN, the *Data Orchestrator* goes through several steps shown in Figure 4.9. It first receives necessary connection information (name, endpoint, access key, and secret key) to connect to it. Once the connection information is given, the *Data Orchestrator* verifies its apparent validity. Once done, the *Data Orchestrator* configures the *mc*, a command-line client with the new connection, and sets up an alias according to the provided name. It then uses the tool to configure the MinIO deployment with a new notification webhook that points back to the *Data Orchestrator*. This allows setting notification configurations on storage buckets and facilitates MinIO to send notifications back to the *Data Orchestrator* when changes occur. After this, the MinIO instance is restarted for the changes to effect fully. *Data Orchestrator* triggers this and waits for it to return to fetch the MinIO instance's updated metadata. The *Data Orchestrator* extracts an internally generated deployment ID from this metadata and the MinIO instance's Simple Queue Service (SQS) Amazon Resource Name (ARN), an identifier required for bucket notification configurations.

Manipulating buckets and objects: One of the goals of *FDN* is to provide a layer of abstraction on top

of the storage deployments and allow users to interact with them in a unified fashion. For this purpose, *FDN* allows for the creation and deletion of storage buckets and the addition, downloading, and deletion of storage objects. All these operations are possible using MinIO Go SDK, using the package's `AddBucket`, `RemoveBucket`, `PutObject`, `Getobject`, `DeleteObject` functions. The *Data Orchestrator* needs to monitor all storage buckets across various clusters. It adds itself as a notification target for each newly created bucket to receive bucket notifications from MinIO whenever a bucket is changed. The *Data Orchestrator*'s API further allows clients to manipulate data objects directly by acting as a go-between and using the MinIO SDK, essentially proxying objects between the client and the MinIO services.

Handling notifications: *Data Orchestrator* has a dedicated API endpoint at the path `/api/notify` that is designed to handle MinIO notifications. The storage deployment that generates the notification sends a JSON object in the request body. This JSON object is parsed to extract the affected object's `key` and the storage deployment's `deployment_id`. The `key` is the object's path, including the bucket in which it is placed. Therefore, it is simple for the program to get both the storage bucket and object names. The `deployment_id`, which matches the database's `minio_deployment_id` column, is used to identify which MinIO deployment sent the event. *Data Orchestrator* then recognizes which object, in which bucket, and on which storage deployment has been affected. Suppose the bucket that originated the event is a master copy. In that case, *Data Orchestrator* mirrors the new contents to all the replica buckets and updates the information in the *FDN Inventory Database*. It also sends a notification to *Courier Control Plane* for reconfiguring the load balancer with the new routes.

Mirroring buckets: *Data Orchestrator* relies on the `mc mirror` command, a feature of the `mc` command-line tool that allows bucket mirroring and read replication between buckets on MinIO storage deployments.

4.2.6 Functions Management

This component is responsible for managing the functions within FDN. The management of functions include creating, deleting and updating of functions. *Functions Management* performs several steps for managing functions and are shown in Figure 4.10. The workflow in Figure 4.10 is divided into two parts:

Function Create/Update: Developers submit their function's code using the *FDN-UI* along with specifying some additional configuration parameters, for example, the memory requirement of the function and the runtime of the function. The input is validated against any wrong specification and once it is validated, the *Functions Management* save the function code within a bucket in the MinIO instance, i.e. *Function Code* in Figure 4.3 and metadata of function in *FDN Inventory database*. Further, *Functions Management* creates a deployment template deployment YAML file for the function and saves it also in *Function Code* MinIO instance. The configuration parameters for the function are passed as environment variables in the template file. The detailed description of the generated template file used for creating a function is shown in Listing A.2 in Appendix A. Based on this deployment template, *Functions Management* uses `kubectl-based create / update` command for creating or updating the deployments on the respective Virtual Kubelet node representing the actual serverless computer cluster. Virtual Kubelet maps create or update commands to the function commands for the respective serverless compute platforms. Lastly, it notifies the *Courier Control Plane* for the function create / update, to update the load balancer configuration.

Function Delete: In this case, developers submit the function delete request from the *FDN-UI*, where the request is validated for right authorization. Once it is validated, *Functions Management* removes the function code from the bucket in *Function Code*, and it's metadata from *FDN Inventory database*. After which, *Functions Management* uses `kubectl-based delete` command for deleting the pod on the respective Virtual Kubelet node, which further initiates the function delete command for the respective serverless compute platform. Lastly, it notifies the *Courier Control Plane* for the function delete, to update the load balancer configuration.

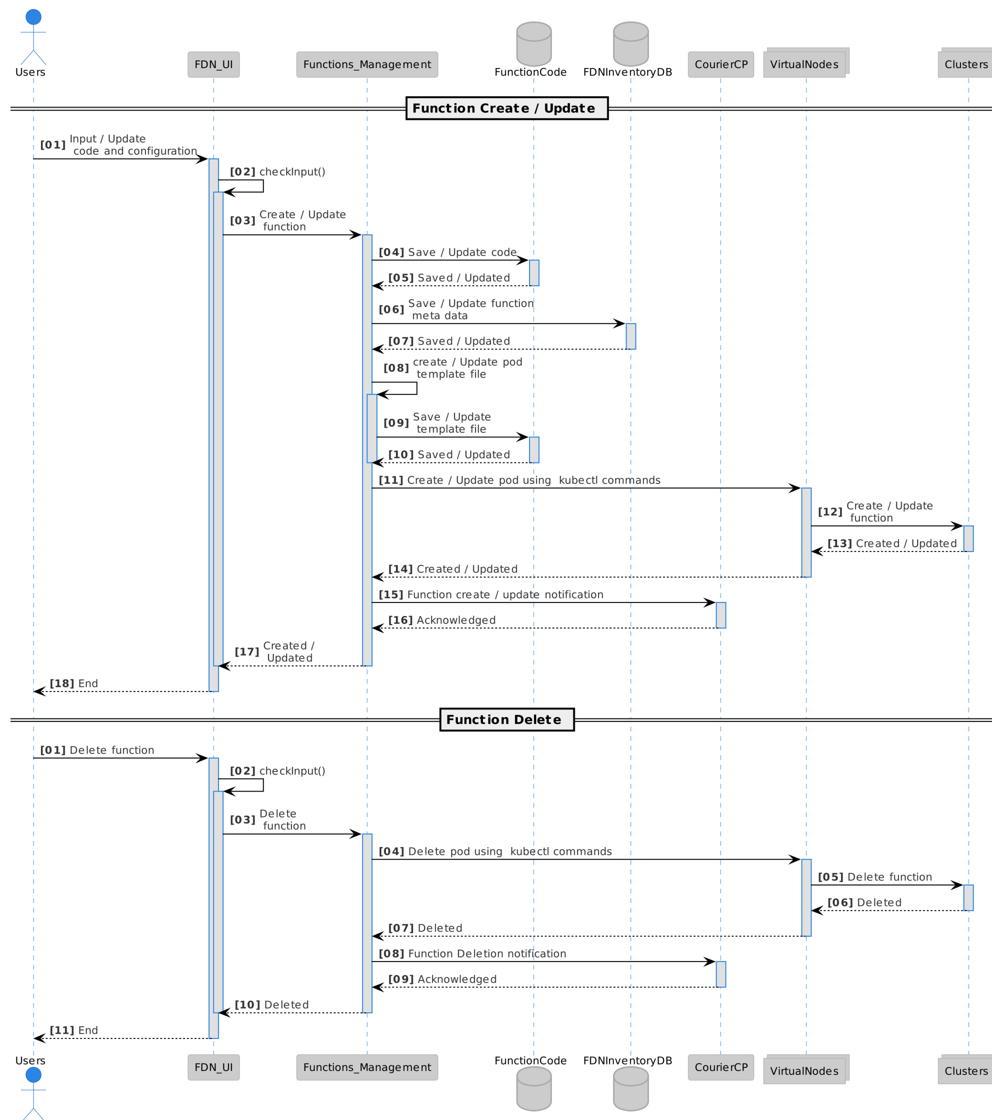


Figure 4.10.: FDN's Function Management workflow showing function create/update and delete.

4.2.7 Behave

Behave represents the behavioral modeling of FaaS functions within FDN based on the monitoring data collected by **FDN-Monitor** for characterization of the FaaS functions. In this regard, we have build two models:

- Functions Performance Model : For automatic end-to-end automatic performance modeling of functions, we develop a python-based tool called **FnCapacitor**. This tool automatically estimates the capacities of FaaS functions within a serverless application under the given SLOs and the memory configuration of the function instance. Furthermore, as part of **FnCapacitor**, we present a novel method which can be used to sandbox individual functions from the serverless application. Currently it supports GCF, and AWS Lambda.
- Functions Interaction Model: We model the serverless applications in the form of function compositions using neural Temporal Point Processes (TPPs). This is developed as part of a tool called

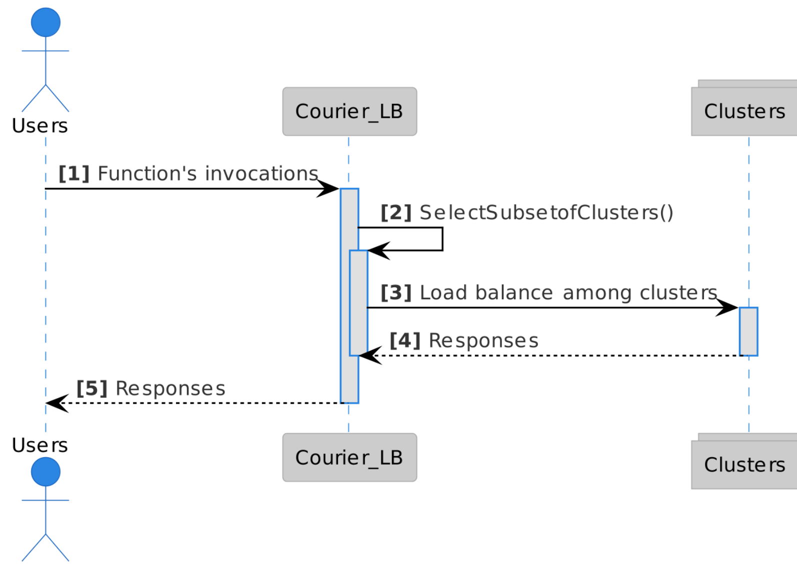


Figure 4.11.: FDN's Function invocation workflow. The user requests are received at the *Courier Load Balancer* which selects a subset of clusters based on the set delivering policy, function name and *X-FDN-Bucket* header. The *Courier Load Balancer* then load balances the invocations across the selected subset of serverless compute clusters based on the set load balancing algorithm.

TppFaaS on top of OpenWhisk. **TppFaaS** uses two neural TPPs: 1) LogNormMix for providing the probability distribution of functions within the composition for the following function invocation, and 2) TruncNorm for predicting a function's invocation time. Such modeling and prediction can avoid cold starts by scaling functions in advance and reducing network load by optimizing the function-server assignment.

This component is described in detail in Chapter 5.

4.2.8 Courier Control Plane and Load Balancer

In order to distribute the load of the incoming invocations among the target serverless compute clusters spread across the edge-cloud continuum, we created **Courier**. **Courier** is responsible for delivering the invocations of the function to the suitable set of clusters. **Courier** mainly consists of two components: 1) *Load Balancer* and 2) *Control Plane*. *Load Balancer*, as the name suggests, is the main entry point for the users and is responsible for load balancing users' functions invocations across multiple serverless compute clusters spread across the edge-cloud continuum based on the set configuration. The *Load Balancer* itself consists of two layers. The first one is the access point from the outside to the FDN, offering an HTTP endpoint. Successively, the requests are dispatched to the second layer using the set function delivery policies (function-awareness and data-awareness). Depending on the policy, a subset of clusters is selected, and a different load balancer is employed. Data-aware delivery takes the bucket name required by the function as the HTTP header parameter. In the second layer of the *Load Balancer*, invocations are load balanced across the selected subset of serverless compute clusters based on a load-balancing algorithm. The *Control Plane* is the brain of the **Courier**. It is responsible for configuring the *Load Balancer* based on various function delivery policies and load balancing algorithms. Figure 4.11 shows the schematic diagram of the steps performed when a function invocation request is received by FDN. This component is described in detail in Chapter 6.

4.2.9 FDN-UI

Though users can interface with FDN through the backend server's API, a user interface developed in ReactJS that allows users to interface with the system easily is also developed. This frontend client is a JavaScript single-page application. It is a static website served by the FDN server next to the backend API, and users can access it through a browser. Upon initial page load, the frontend application sends an HTTP GET request to the FDN's various components' to retrieve the current status of FDN and settings stored in the database. Once the data is retrieved and parsed, the application allows users to interact with up-to-date data. The application is split into different pages dedicated to the different entities in the FDN. Users can access each page by clicking on links displayed throughout the client application or navigating to their URL in the browser. The JavaScript application parses the URL and uses it to determine the appropriate view to display. The screenshots of the UI are presented in Figure A.2 in Appendix A.

4.3 Summary

In this chapter, we described an extension to the concept of FaaS as a programming interface for serverless computing across the edge-cloud continuum called **Function Delivery Network (FDN)**. FDN provides seamless integration across the edge-cloud continuum by allowing the user to deploy and invoke the functions across heterogeneous serverless compute clusters in the continuum. FDN provides **Function-Delivery-as-a-Service (FDaaS)**, which can deliver user workload functions invocations to a subset of serverless compute clusters spread across the continuum based on : 1) function-awareness, and 2) data-awareness. The invocations are load balanced across the selected subset of clusters based on the set load balancing algorithm. In order to integrate several serverless compute clusters, we presented our custom *Virtual Kubelet* provider called *FDN-provider*. *FDN-provider* acts as federation for multiple serverless compute clusters. Every *Virtual Kubelet* node created using *FDN-provider* acts as a proxy to a serverless compute cluster.

We introduced various components of FDN, starting with the serverless compute clusters that are part of FDN (§4.2.1). We presented three clusters: edge clusters, private cloud clusters, and public cloud clusters. We further described the automation used in this work for creating those clusters using Ansible and Terraform (§4.4). For collecting various metrics from serverless clusters using different serverless platforms, we presented a tool called *FDN-Monitor* (§4.2.2). It is deployed as a sidecar to the virtual nodes representing the serverless compute clusters. We then described about the database model used in the FDN (§4.2.3). *Clusters Management* component within FDN is responsible for managing the clusters spread across the continuum (§4.2.4). We explained about the *Data Orchestrator* within FDN, responsible for managing the data across the MinIO on various clusters (§4.2.5). For managing the functions across the continuum in FDN, we created a component called *Functions Management* (§4.2.6). We gave a high-level overview of the *Behave* and *Courier* components within FDN, but they are presented in detail under separate chapters. Lastly, we presented *FDN-UI*, through which users can interface with FDN.

5

Behave: Behavioral Modeling of FaaS Functions in FDN

“The secret of getting ahead is getting started."

— Mark Twain

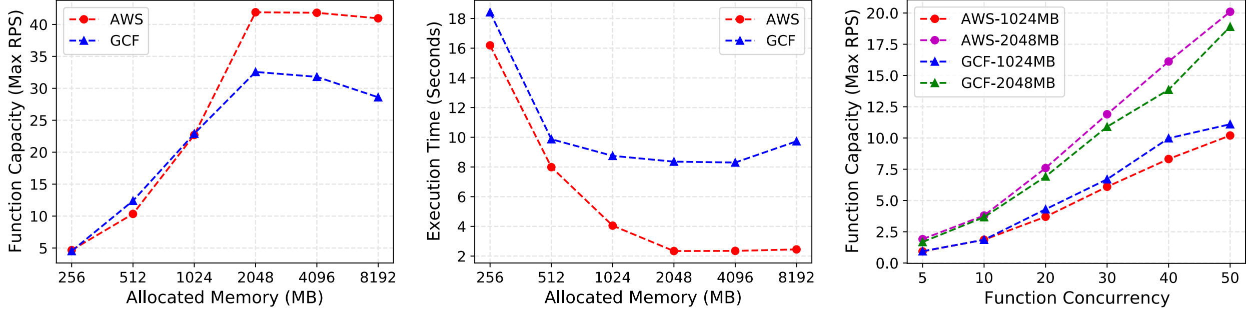
This chapter presents two behavioral models based on the monitoring data collected by *FDN-Monitor* for characterization of the FaaS functions: 1) Functions Performance Model (§5.1), and 2) Function Interaction Model (§5.2).

5.1 Functions Performance Model

The Function Performance Model captures the performance with respect to time for certain combinations of resources, such as the number of cores, the network bandwidth, the memory size, and the I/O bandwidth. Since serverless computing environments abstract the underlying system infrastructure configurations away from the users, most public cloud providers in their serverless compute platforms allow users to configure only a small set of configuration parameters. These parameters include memory allocation and the maximum number of function instances, called concurrency [10, 79, 44]. Moreover, cloud providers speedup function execution when a higher memory is configured [179]. Furthermore, the heterogeneity in the underlying nodes can lead to variations in the execution time of the FaaS functions [302]. Therefore, defining and building a *Functions Performance Model* is challenging. Towards this, we first introduce the concept of *Function Capacity (FC)* (§5.1.1) used for defining the *Functions Performance Model*, and then the tool *FnCapacitor* (§5.1.2) for building it.

5.1.1 Function Capacity (FC)

We define the *Function Capacity (FC)* as the maximal number of concurrent invocations that a FaaS function, when deployed on a serverless compute platform with a certain memory configuration and fixed max-



(a) Effect of memory on the FC with fixed function concurrency of `100`.

(b) Effect of memory on the FC with fixed function concurrency of `100`.

(c) Function concurrency effect on the FC with fixed memory.

Figure 5.1.: Function Capacitys (FCs) (maximum requests per second) variation with different memory configurations, and function concurrency for AWS Lambda and GCF.

imum function instances, can serve within a time interval without violating the SLOs. In this work, we consider the 95th percentile execution time of a FaaS function as the SLO. Ideally, if an instance i serves n_i^t number of invocations within a time interval t and C is the function concurrency for the serverless compute platform, then the $FC(f)$, where f is any function, can be calculated by the equation (5.1).

$$FC(f) = n_i^t \times C \tag{5.1}$$

However, in practice, many other factors affect the performance and Function Capacitys (FCs) of FaaS functions [302]. To highlight the effects of various parameters on the performance and the FCs of FaaS functions, we deployed a compute-intensive (calculates prime numbers till 10000000) serverless function written in Python on AWS Lambda and GCF [79]. We fixed the function's 95th percentile execution time to `20s`. Figure 5.1a shows that the FC first increases with varying memory sizes upto a certain point (`2048MB`), and after that, it becomes constant when the function concurrency is fixed to `100` for both the platforms. The same Figure 5.1a also shows the variation in FC with the platforms. The variation in the system resources causes differences in performance between the same function deployments for the same serverless compute platform. Figure 5.1b shows the execution time of corresponding runs, and one can see that it decreases with the increase in memory, and after a point (`2048MB`), it also becomes constant. Lastly, Figure 5.1c shows the linear increase in FC with the increase in function concurrency, keeping the allocated memory fixed.

5.1.2 FnCapacitor

The examples described in §5.1.1 highlights some factors that can affect the performance and the FCs. However, many other factors such as cold starts, I/O and network conditions, type of container runtimes, and co-location with other functions affect the performance and FCs which the users are not aware of [302]. Additionally, the dependencies between the functions within a serverless application can affect the FCs. To this end, we develop **FnCapacitor**, a tool that can estimate the FCs of the functions adhering to the given SLOs, the specified memory configurations and function concurrency [148]. In this section, we briefly overview its working and introduce its two important components.

FnCapacitor is developed in python. Given the SLO requirements, it is responsible for estimating the FCs of FaaS functions at different deployment configurations (memory allocation and function concurrency) by

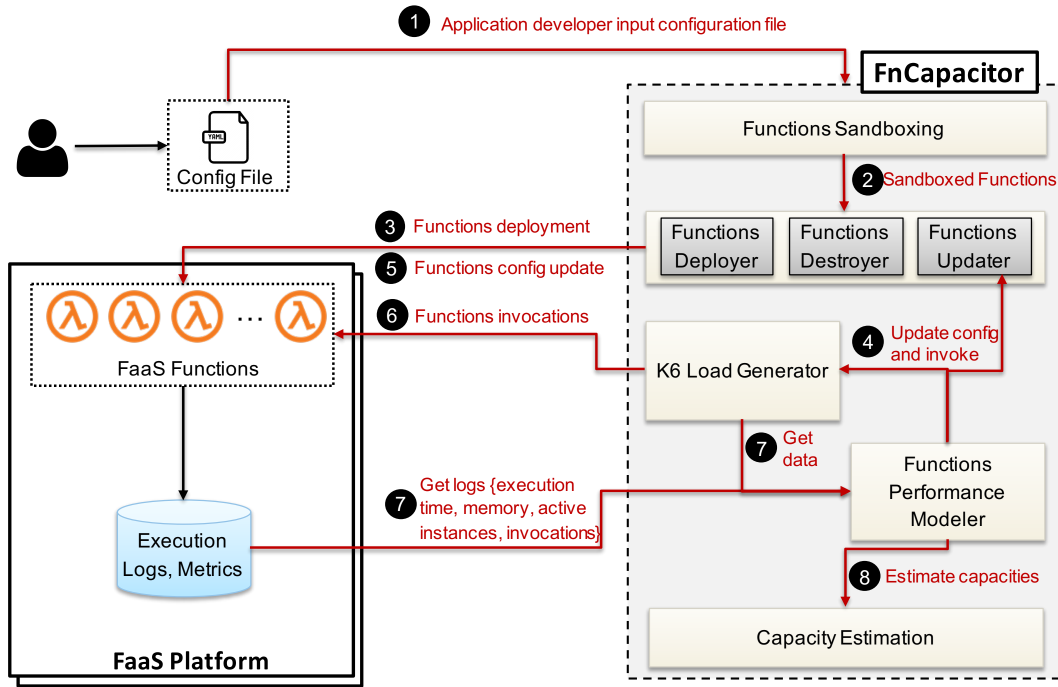


Figure 5.2.: High-level architecture of the *FnCapacitor* and the interaction between its components in a general use case [148]. *FnCapacitor* takes a YAML file as input, and the individual functions from the given application are segregated. These sandboxed functions are then deployed on a serverless compute platform. After the deployment, *FnCapacitor* generates a user workload and repeatedly changes the functions configurations to collect data. The collected metrics data is used for creating the function performance models and are then used for estimating the FCs for different deployment configurations.

conducting a limited set of load tests followed by building machine learning models on the acquired performance data. Figure 5.2 provides an overview of the high-level architecture of *FnCapacitor*. It takes a YAML file as input that specifies the initial serverless compute platform configuration parameters (minimum memory allocation and functions timeout), serverless application to be deployed, and configuration parameters for the load generator and the modeling (step ❶). Since a serverless application consists of multiple functions, in the next step, the individual functions from the given application are segregated (step ❷). These sandboxed functions are then deployed on the serverless compute platform with initial configurations (step ❸). After the deployment, *FnCapacitor* repeatedly change the functions configurations (steps ❹ - ❺) and generates a user workload to the function's API endpoint (step ❻) for collecting various monitoring metrics data (§5.1.3.1). The collected metrics data is used for creating the function performance models (step ❼). The created function models are then used for estimating the FCs for different deployment configurations (step ❽).

Functions Sandboxing: A serverless application consists of multiple functions, and the performance of one function could affect the others depending on it. Therefore to measure the pure performance of the functions, i.e., where their performance is not affected by others, we sandbox the individual functions through a mockup of their neighbors. It isolates each function and substitutes its direct neighbors with dummy functions accepting the requests and sending the responses in the same format, but without any additional processing, allowing us to measure the pure performance of only that function and build models using this data. Firstly, each function calls to other functions are replaced with calls to a *proxy-function*. This

proxy-function serves as an intermediator between the sandboxed function and other functions and takes the originally called function names from the sandboxed function and the payload as the input. Thus, every invocation to other functions goes through this *proxy-function*, and this dummy *proxy-function* will invoke the following functions based on the input received. At the same time, copies of the requests and responses are stored in the FnCapacitor's MongoDB database for creating function mockups. It is to be noted that BaaS services such as database, storage, and queues, are out of the scope of this work for sandboxing as it is assumed that these BaaS services provide high scalability and serve the user requests within the defined SLOs. Each function receives its own sandboxed deployment, where mockup functions replace the direct neighbors. These mockup functions will respond with the response stored in MongoDB. As a result, the time the dependent functions take to respond becomes negligible, allowing the building of a relatively pure performance model of the functions within a serverless application.

Performance Model Builder and Capacity Estimation: It is responsible for analyzing the correlation between the different monitoring metrics data and the deployment configurations. It uses the collected data to create models of the functions and estimate their FCs. Modeling approaches used in this work are categorized under two categories:

- **Statistical Approaches**: We use linear, polynomial, ridge, and random forest regression to model the relationship.
- **Deep Neural Networks (DNNs)**: DNN model generator consists of a *Normaliser* and a DNN. The *Normaliser* normalizes the numeric columns in the dataset using l2-normalization to use a standard scale without distorting differences in the ranges of values or losing information. The DNN is built with a sequential model consisting of an input layer, two hidden layers, and an output layer. Both the hidden layers are dense with `64` units and a *Rectified Linear Unit (ReLU)* activation function. The model generator compiles the DNN model with 'mean_absolute_error' as the loss function and 'Adam' as the optimizer.

The collected data is pre-processed by removing outliers and dividing the data into training and test set. Following this, different models, i.e., statistical and DNNs, are trained on the partitioned training data set. Due to the sparse training data, k-fold cross-validation (in our case, k=6) is used for training [316].

5.1.3 Experimental Configuration

In this work, we have fixed the total duration of a test to `30` minutes for the deployed serverless application. A test consists of the memory allocation configurations: `<256MiB, 512MiB, 1GiB, 2GiB and 4GiB>` and function concurrency: `<10, 20, 30, 40 and 50>`. AWS Lambda functions are deployed in `europe-central1` region and GCF functions are deployed in `eu-west3` region, and the number of Virtual Users (VUs) during the load generation from *k6* [306] are varied from `5` to `500` depending on the amount of requests the functions can serve. As a result, for each function, 5 (total memory configurations) $\times$ 5 (function concurrency configurations) = 25 tests were conducted. To evaluate our approach, we partition the collected data into training and test set (`33`% of the total data). We used part of the training data set as a validation set (§5.1.2) for selecting the hyperparameters of the different models. We select the hyperparameters through an exhaustive grid search. For the DNN model, we use a `12-layer` fully connected neural network architecture, with each layer having `64` units. We use the ReLU as the activation function, as usage of ReLU has been proven to lead to faster convergence [163]. We also implemented early stopping with patience five to ensure no over-fitting on the training data [67].

To investigate the performance of each deployment configuration, we used the benchmarks described in §9.1.1 and the application described in §9.1.2 and shown in Figure 9.1.

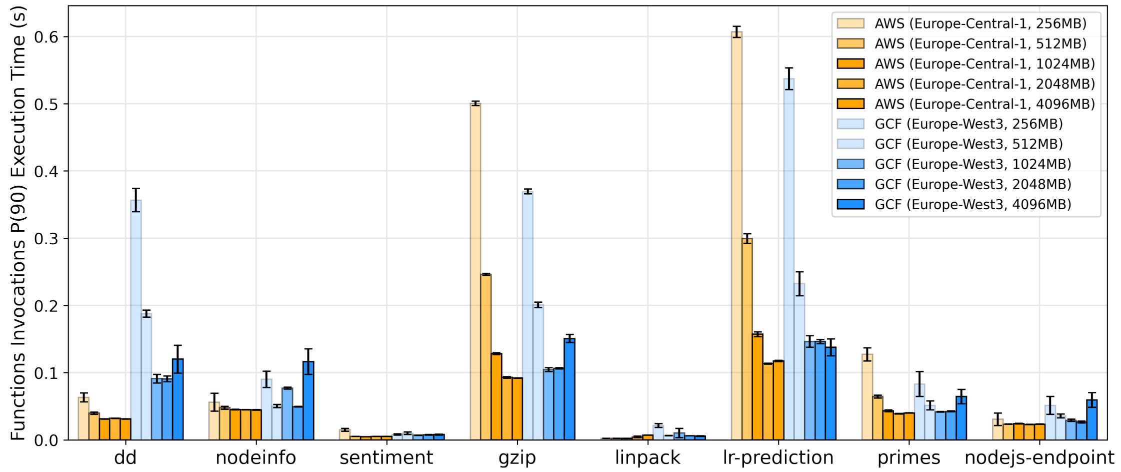


Figure 5.3.: `execution_durations` of the sandboxed functions when executed with a load of 50 Requests per Second (RPS) and no limit on the `function_concurrency`.

5.1.3.1 Monitoring Metrics

We extracted the following monitoring metrics from the GCF and AWS Lambda with sampling rate of `60s`:

- `concurrent_instances`: The number of active concurrent function instances.
- `invocations`: The number of times the function code is executed.
- `execution_duration`: The amount of time a function code spends in processing an event.
- `memory_usage`: Function's maximum memory usage during execution.
- `allocated_memory`: The amount of memory allocated to the function.
- `function_concurrency`: The maximum number of concurrent instances allowed for processing events.

5.1.4 Experimental Results

In this section, we first describe the impact of heterogeneity in the memory allocations on function's `execution_duration` and `concurrent_instances`, and then the effect of `function_concurrency` on the FCs for both the serverless compute platforms and on different functions. Following this, we present the results of FCs estimation using the different modeling approaches.

5.1.4.1 Memory Effect on Function Execution Duration

Figure 5.3 shows the `execution_durations` of the load testing of 50 invocations per second on the application when the functions are sandboxed and deployed with five different memory configurations on GCF and AWS Lambda platforms. We observe the following:

Decrease in `execution_duration` with the increase in memory and becoming constant: From Figure 5.3, we can see that for most of the functions and across two serverless compute platforms, there is a general trend of decrease in `execution_duration` with the increase in memory, and after a certain point (`2048MB` memory configuration), either it becomes constant (for all functions running on AWS, and `lr-regression`,

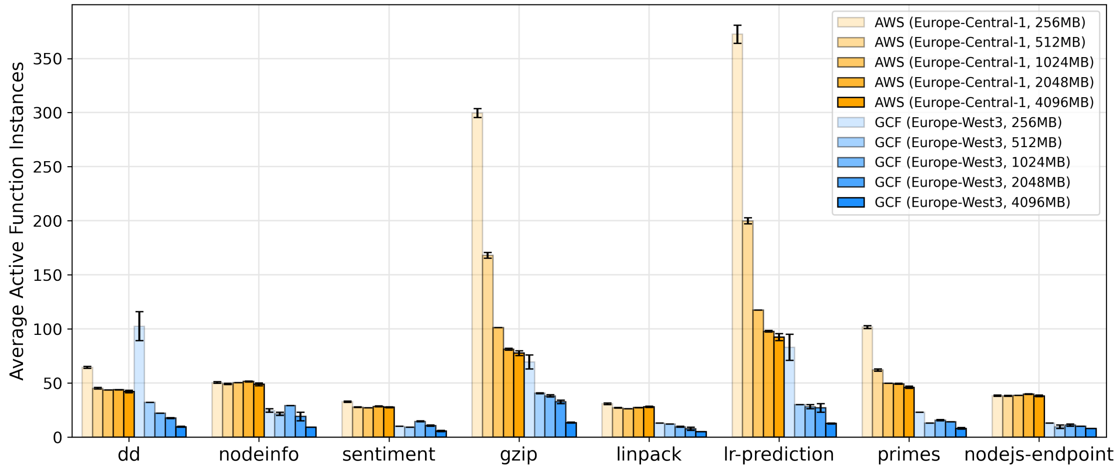


Figure 5.4.: `concurrent_instances` of the sandboxed functions for handling the load of 50 RPS with five different memory configurations and no limit on the `function_concurrency`.

`sentiment-analysis`, and `linpack` on GCF) or increases (for all other functions on GCF). This behavior can be attributed to an increase of $2x$ in the number of allocated clock cycles for a memory configuration of 4096MB compared to 2048MB [120].

In general, AWS Lambda has a lower execution duration for most of the functions at all memory configurations as compared to GCF: We can observe from the Figure 5.3 that, in most of the functions except the three compute-intensive functions at lower memory configurations ($256MB$ and $512MB$), AWS Lambda process function events faster than the GCF. For example, for `dd` microbenchmark at $256MB$ configuration, AWS Lambda takes $5.2x$ less time than the GCF at the same memory allocation. `nodeinfo`, `sentiment-analysis`, `linpack`, and `nodejs-endpoint` took almost the same time across different memory configurations and serverless compute platforms.

5.1.4.2 Memory Effect on Function's Concurrent Instances

Figure 5.4 shows the `concurrent_instances` per function in the serverless application when it is load tested with a load of 50 invocations per second on GCF and AWS Lambda platform for five different memory configurations. We observe the following:

AWS Lambda creates more `concurrent_instances` as compared to GCF: From Figure 5.4, we can see that, for most of the functions and across different memory configurations, AWS creates a higher number of `concurrent_instances` as compared to GCF for handling the same amount of load.

Decrease in number of `concurrent_instances` with the increase in memory configuration: As the memory is increased for each function, the number of `concurrent_instances` for both the serverless compute platforms either remains constant or decreases. This trend can be attributed to the fact that a higher resource instance can serve the requests faster and hence can process more requests per unit time. Therefore, fewer instances are required to handle the same load when allocated with lower memory configurations.

Slow-processing functions are scaled to higher number of `concurrent_instances` to match up with the fast-processing functions : From Figure 5.3, we can observe that, `lr-regression` and `gzip-compression` have the highest `execution_time` as compared to the other functions. When observing the number of `concurrent_instances` for those two functions in Figure 5.4, we can see that they are also the highest. It concludes that the compute-intensive (slow-processing) functions require higher scaling to match up with the

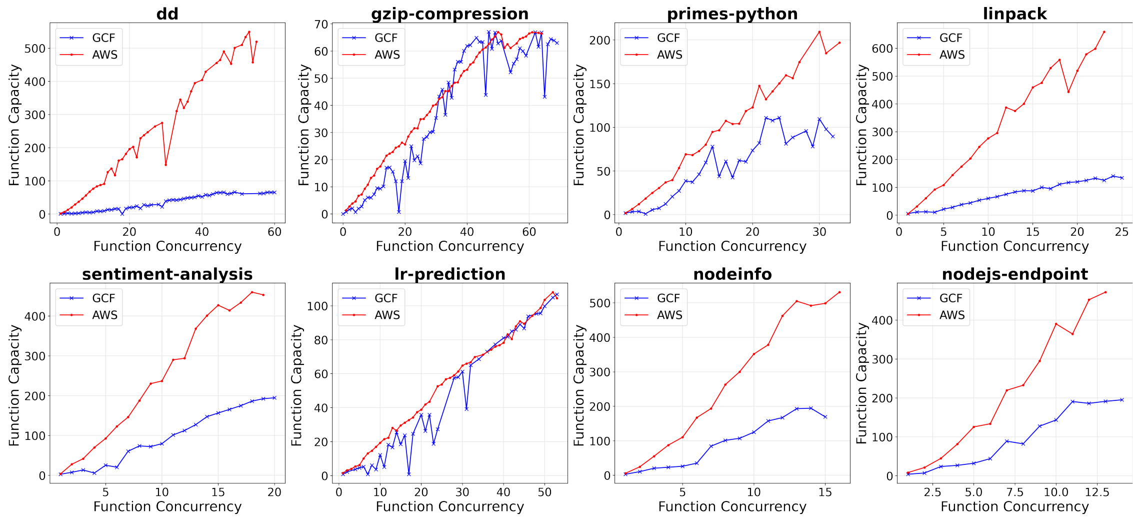


Figure 5.5.: FC of the functions when deployed on the two serverless compute platforms for different `function_concurrency` with memory configuration fixed to `256`MB.

other fast-processing functions for handling the same workload. Such visualization can also be used to understand the bottleneck function in the serverless application; for example, in our case, it is `lr-regression`.

5.1.4.3 Effect of Function Concurrency on the FC

Figure 5.5 shows the actual capacity measurements for the two serverless compute platforms for different `function_concurrency` configurations, with memory configuration fixed to `256`MB for all the functions. The capacities are the average of the five runs for both serverless compute platforms. In general, it can be inferred that for most of the functions, FCs vary linearly with the `function_concurrency` for both the serverless compute platforms. Also, a single instance of AWS Lambda can process a higher number of requests than the single instance of GCF. However, for the two compute-intensive functions namely: `gzip-compression`, and `ml-lr-prediction` we see a similar FCs for both the platforms. In case of GCF, for simple web-based functions : `sentiment-analysis`, `nodeinfo`, and `nodejs-endpoint` the linear increasing slope is not constant. From Figure 5.5, one can see that, for the three FaaS functions, the linear slope changed after `function_concurrency` of `6`. This means that, after the `function_concurrency` of `6`, each instance can process more requests. Generally, the trend is linear for all other functions, and serverless compute platforms. However, they are not exactly following the ideal lines. Therefore, one needs modeling approaches for the estimation of FCs for both the serverless compute platforms.

5.1.4.4 Function Capacity Estimation

On analyzing the impact of varying memory configurations on the performance of the different FaaS functions, we use the metrics <concurrent_instances, execution_duration, allocated_memory, memory_usage, function_concurrency> obtained from the collected load test data as input parameters for the different models (§5.1.2). For a given set of input parameters, all models predict `<function invocations>` which is equivalent to the FC. The collected data is split between training and test set, with 33% of the data being used as the test set and the rest for training.

Table 5.1.: Comparison of accuracy results (R^2 score) for estimated FCs for the different approaches.

Function	LR		PLR		RR		RFR		DNN	
	GCF	AWS	GCF	AWS	GCF	AWS	GCF	AWS	GCF	AWS
dd	81.9	98.1	88.27	97.7	82.2	98.0	88.5	98.1	91.1	98.2
gzip	83.5	91.4	89.8	94.8	83.6	91.4	93.0	94.6	93.6	94.9
primes	75.8	95.4	78.6	95.1	76.4	95.6	83.1	96.7	85.0	96.5
linpack	86.3	58.7	87.7	75.9	86.4	76.3	88.5	87.2	88.8	92.4
sentiment	65.6	33.6	72.9	92.9	52.2	24.9	76.0	97.4	74.4	96.2
lr-pred.	90.9	99.4	93.0	99.5	90.7	99.4	95.4	98.7	94.7	99.5
nodeinfo	80.6	87.5	88.4	87.6	79.6	87.9	89.6	88.2	90.2	87.6
endpoint	77.2	36.5	80.8	67.9	76.3	35.7	82.8	81.0	77.8	80.2

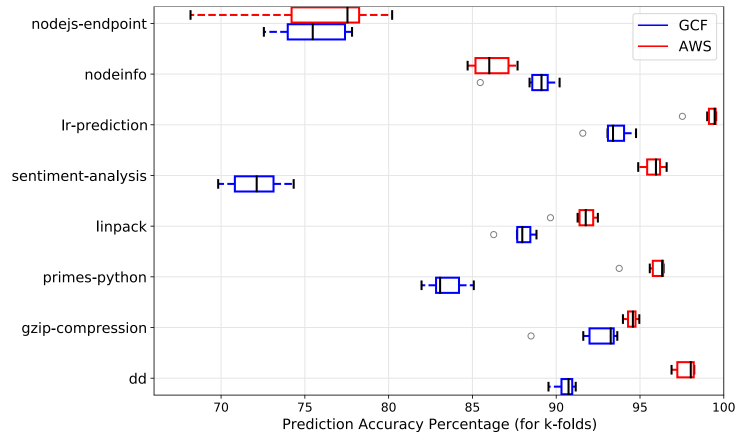


Figure 5.6.: Box plot showing the prediction accuracy on the test data across k-folds using DNN model for both serverless compute platforms.

Table 5.1 shows the comparison of the accuracy results for estimated FCs for the different modeling approaches (§5.1.2) with the best ones underlined for both the serverless compute platforms. To determine the formulated models' accuracy, we use the R^2 score [290].

In general, it was found that the accuracy measurements for FC estimation for AWS Lambda are higher than the GCF for most of the FaaS functions, since AWS Lambda exhibits more linear behavior as compared to GCF (Figure 5.5). **Linear Regression (LR)** leads to best results when the parameters are linearly correlated to the FC. For most FaaS functions, the parameters are linearly correlated to FC, leading to an accuracy value greater than 80%. For both the serverless compute platforms, the accuracy for the `nodejs-endpoint`, and `sentiment-analysis` FaaS functions is comparatively less compared to other FaaS functions since most parameters in them are non-linearly correlated with FC. **Polynomial Linear Regression (PLR)** leads to highly accurate results for most of the function types due to its ability to model non-linear relations among the parameters. **Ridge Regression (RR)** produced approximately the same results as linear regression and worked well for certain function types. On the other hand, **Random Forest Regression (RFR)** can provide the best results among the statistical approaches.

The **DNN** method outperformed all the statistical approaches for most of the FaaS functions, since it is capable of modeling the linear and non-linear correlation between the parameters. For most function types, the FC estimation accuracy is approximately above 75%. In Figure 5.6, we show the prediction accuracy percentage for k-folds using the box plot for both serverless compute platforms and all the FaaS functions.

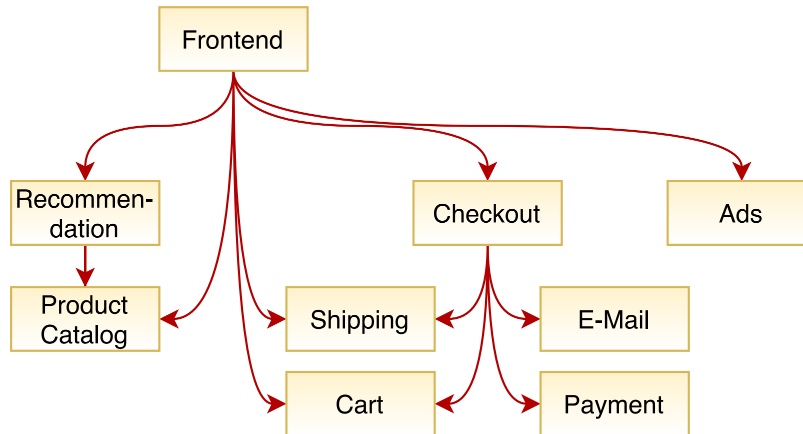


Figure 5.7.: A webshop implemented as a composition of FaaS functions [122].

5.2 Functions Interaction Model

A serverless application is often constructed as a composition of multiple functions that abstract some business process [52]. Therefore *Functions Interaction Model* characterizes the producer-consumer interactions of functions within a serverless application. The interaction might, for example, suggest packaging functions together to reduce communication costs. An example of such a composition can be seen in Figure 5.7, in which multiple FaaS functions implement a webshop [122]. Each function fulfills a simple modular logic, with the interaction between functions enabling a complex program. Orchestration tools such as AWS Step Functions [261], Azure Durable Functions [193], or OpenWhisk's Composer [223] facilitate building such compositions. These provide constructs to compose the functions into a control flow known from any imperative programming language. A developer can arrange the functions sequentially, in parallel, or in loops and integrate branching and conditional logic. In addition, the function orchestrator performs other important tasks such as state management, i.e., storing the data communicated between functions, error handling, real-time monitoring, logging, and much more [262].

Since FaaS follows an event-based execution model, we can model the events triggering the functions using Temporal Point Processes (TPPs). A Temporal Point Process (TPP) is a probability distribution over sequences of instantaneous points in time, denoted as events, of variable length in an interval $[0,T]$ [270]. Therefore, TPPs are perfect for modeling interactions of functions within a serverless application by representing an executed function composition by a sequence of events. Such modeling of FaaS function compositions and then a prediction can avoid cold starts by scaling functions in advance and reducing network load by optimizing the function-server assignment. Furthermore, it can also help in optimizing the data-function placement. In this regard, we developed a python-based tool called *TppFaaS* on top of OpenWhisk for modeling serverless functions invocations via TPPs [279].

In this section, we first give a brief overview of TPPs and introduce the neural TPPs used in this work in §5.2.1, and then presents a high-level overview of the developed tool *TppFaaS* in §5.2.2.

5.2.1 Temporal Point Processes (TPPs)

A TPP is a probability distribution over sequences of instantaneous points in time, denoted as events, of variable length in an interval $[0,T]$ [270]. These events are discrete in continuous time. Discrete means

Table 5.2.: Symbols and their definitions used in the context of building *Functions Interaction Model*

Symbol	Interpretation
x	sequence of events
N	number of events in the given event sequence x
$t_1,\ldots,t_N$	event's occurrence times
$m_1,\ldots,m_N$	event types (or marks as referred in the literature) at different times
τ_i	inter event time $(t_i - t_{i-1})$
$\mathscr{H}(t)$	the history of past events for a given event sequence x
$f_i^*(t_i)$	conditional probability density function for modeling the event times of a Temporal Point Process (TPP) model
$F_i^*(t_i)$	cumulative distribution function for modeling the event times of a TPP model
$S_i^*(t_i)$	complementary cumulative distribution function also known as survival function for modeling the event times of a TPP model
$\lambda^*(t)$	conditional intensity function for modeling the event times of a TPP model
$\phi_i^*(t)$	*hazard function* for characterizing a TPP model
μ	constant event rate in homogenous Poisson process
$\kappa(t)$	kernel function in the Hawkes process for modeling the dependence on previous events

that events can be categorized into classes, often referred to as event type or mark in the literature [269]. A realization of a marked TPP model can be represented as an event sequence $x = \{(t_1,m_1),\ldots,(t_N,m_N)\}$, where $0 < t_1 < \cdots < t_N < T$ represents event's time (see Table 5.2) with N being the number of events and is itself a random variable, and m_i represents an event type or a mark. In most applications, the marks $(m_1,\ldots,m_N)$ are categorical, such that $m_i = \{1,\ldots,K\}$, although other representations are possible for this. Furthermore, a TPP can also be represented by a list of strictly positive inter-event times $\tau_i = t_i - t_{i-1} \in \mathbb{R}_+$, where $t_0 = 0$ and $t_{N+1} = T$. Both notations are equivalent and can be replaced with each other as desired. Finally, $\mathscr{H}(t) = \{(t_j,m_j)|t_j < t\}$ defines the history of past events for a given event sequence x.

Each event time t_i is a random variable, which is modeled in an autoregressive fashion by the TPP model, i.e., conditioned on past events defined by the history $\mathscr{H}(t_i) = \{t_1,\ldots,t_{i-1}\}$. Modeling t_i is equivalent to modeling the inter-event time τ_i for a given $\mathscr{H}(t_i) = \mathscr{H}(t_{i-1} + \tau_i)$. For the sake of simplicity, in the following subsections we consider an unmarked TPP such that $x = \{t_1,\ldots,t_N\}$. The modeled distribution of t_i and τ_i, respectively, can be characterized for a given $\mathscr{H}(t_i)$ by one of the following three functions, also illustrated in Figure 5.8:

- The conditional probability density function $f_i^*(t_i) = f_i(t_i|\mathscr{H}(t_i))$ determines the probability that the next event for a given history $\mathscr{H}(t_i)$ occurs in the interval $[t_i, t_i + dt)$. Similarly, the conditional density function $f_i^*(\tau_i) = f_i(\tau_i|\mathscr{H}(t_i))$ defines the probability, that the time until the next event for a given history $\mathscr{H}(t_i)$ is within the interval $[\tau_i, \tau_i + d\tau)$.
- The cumulative distribution function $F_i^*(t_i) = F_i(t_i|\mathscr{H}(t_i)) = \int_{t_{i-1}}^{t_i} f_i^*(u)\,du$ determines the probability that the next event for a given history $\mathscr{H}(t_i)$ occurs before t_i. Similarly, the cumulative distribution function $F_i^*(\tau_i) = F_i(\tau_i|\mathscr{H}(t_i)) = \int_0^{\tau_i} f_i^*(t_{i-1} + u)\,du$ is the probability that the time to the next event for a given history $\mathscr{H}(t_i)$ is less than τ_i.
- The complementary cumulative distribution function $S_i^*(t_i) = S_i(t_i|\mathscr{H}(t_i)) = 1 - F_i^*(t_i) = \int_{t_i}^{\infty} f_i^*(u)\,du$, also called survival function, defines the probability that the next event for a given history $\mathscr{H}(t_i)$

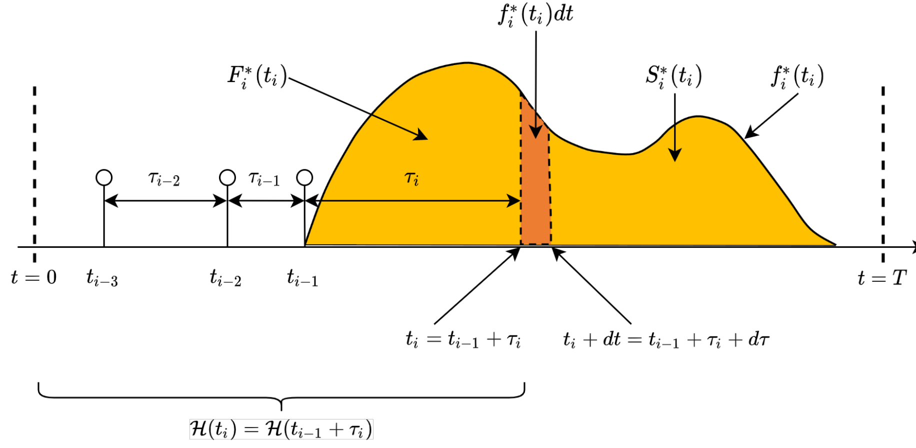


Figure 5.8.: The conditional probability density function $f_i^*(t_i)$, the cumulative distribution function $F_i^*(t_i)$, and the survival function $S_i^*(t_i)$ model the time of the next event t_i for a given event history $\mathscr{H}(t_i)$ for a TPP model [88].

occurs after t_i. Similarly, the complementary cumulative distribution function $S_i^*(\tau_i) = S_i(\tau_i|\mathscr{H}(t_i)) = 1 - F_i^*(\tau_i) = \int_{\tau_i}^{\infty} f_i^*(t_{i-1}+u)\,du$ is the probability that the time to the next event for a given history $\mathscr{H}(t_i)$ is greater than τ_i [88] [270].

Any of the functions f_i^*, F_i^*, and S_i^* can be used to model the distribution of t_i or τ_i. If one of the functions is known, the other two can be derived from it [268]. Many other functions can be used to model the distribution of t_i or τ_i, but a prominent one from the literature is the conditional intensity function $\lambda^*(t)$, which is often used in the literature to describe TPP models.

For interpretation of the conditional intensity function, we consider a representation of the TPP model in which it is defined as a counting process $N(t)$, counting the number of events up to time t. For an infinitesimally time interval dt, it holds that $dN(t) = N(t+dt) - N(t) \in \{0,1\}$, meaning that at most one event can occur in $[t, t+dt)$ [88]. From these follows:

$$\begin{aligned}\mathbb{E}[dN(t)|\mathscr{H}(t)] &= 1 * \mathbb{P}(\text{next event in } [t,t+dt)|\mathscr{H}(t)) \\ &\quad + 0 * \mathbb{P}(\text{no event in } [t,t+dt)|\mathscr{H}(t)) \\ &= \lambda^*(t)dt.\end{aligned} \tag{5.2}$$

If the equation (5.2) is rearranged, the result is equation (5.3).

$$\lambda^*(t) = \lim_{dt \to 0} \frac{\mathbb{E}[dN(t)|\mathscr{H}(t)]}{dt}, \tag{5.3}$$

From equation (5.3), we derive that the conditional intensity function specifies the expected number of events per unit time [268], that is, the frequency rate per unit time, i.e., $\lambda^*(t) = events/second$. The intuitive interpretation facilitates the construction of TPP models with desired characteristics by specifying the functional form of $\lambda^*(t)$. When choosing the functional form of $\lambda^*(t)$, the only constraint is that for any t and $\mathscr{H}(t)$, the two terms $\lambda^*(t) \geq 0$ and $\int_t^{\infty} \lambda^*(u)\,du = \infty$ must be satisfied. In contrast, the conditional probability density function $f_i^*(t)$ must be specified as a valid probability distribution, such that $\int_{t_{i-1}}^{\infty} f_i^*(u)\,du = 1$ is satisfied [88, 268].

If the conditional intensity function $\lambda^*(t)$ is given, the conditional probability density function $f_i^*(t)$ can be derived from it. From the definition of the survival function $S_i^*(t)$ we know that $S_i^*(t) = 1 - F_i^*(t)$, thus

$$\begin{aligned} \frac{dS_i^*(t)}{dt} &= \frac{d}{dt}(1 - F_i^*(t)) \\ \iff -\frac{dS_i^*(t)}{dt} &= f_i^*(t). \end{aligned} \tag{5.4}$$

Plugging equation (5.4) into (8.1) then yields

$$\lambda^*(t) = \frac{f_i^*(t)}{S_i^*(t)} = -\frac{1}{S_i^*(t)} \frac{dS_i^*(t)}{dt} = -\frac{d \log S_i^*(t)}{dt}. \tag{5.5}$$

The integration of both sides of equation (5.5) leads to

$$\begin{aligned} \log S_i^*(t) &= -\int_{t_{i-1}}^{t} \lambda^*(u)\,du \\ \iff S_i^*(t) &= \exp(-\int_{t_{i-1}}^{t} \lambda^*(u)\,du). \end{aligned} \tag{5.6}$$

The derived equation for the survival function from (5.6) is plugged into (8.1), leading eventually to the formula for the conditional probability density function [88]:

$$f_i^*(t) = \lambda^*(t) \exp(-\int_{t_{i-1}}^{t} \lambda^*(u)\,du). \tag{5.7}$$

Furthermore, we introduce the *hazard function* $\phi_i^*(t) = \phi_i(t|\mathscr{H}(t))$, another function to characterize a TPP and which is related to the conditional intensity function $\lambda^*(t)$ [270, 268]. While $\lambda^*(t)$ describes the global intensity in the time interval $[0,T]$, the hazard function $\phi_i^*(t)$ is limited to the time interval between two events $(t_{i-1}, t_i]$, which is why the index i is required. That is, for a sequence of N events, we obtain the global intensity $\lambda^*(t)$ by concatenating the hazard functions $\phi_1^*, \phi_2^*, \ldots, \phi_{N+1}^*$, i.e.,

$$\lambda^*(t) = \begin{cases} \phi_1^*(t) & \text{if } 0 \leq t \leq t_1 \\ \phi_2^*(t - t_1) & \text{if } t_1 < t \leq t_2 \\ \cdots & \\ \phi_{N+1}^*(t - t_N) & \text{if } t_N < t \leq T \end{cases}. \tag{5.8}$$

5.2.1.1 Neural Temporal Point Processes Models

Neural TPP models autoregressively predict the time t_i and mark m_i of the next event by conditioning the prediction on the history of past events $\mathscr{H}(t_i)$. In [270], the authors partition the prediction process of neural TPP models (shown in Figure 5.9) into the following three steps:

1. Each event (t_i, m_i) is mapped to a feature vector $\boldsymbol{y}_i$.
2. The history $\mathscr{H}(t_i)$ is encoded by the history embedding vector $\boldsymbol{h}_i$, which is computed by sequentially feeding $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{i-1}$ into an RNN.
3. The conditional distribution over the next event $P_i(t_i, m_i|\mathscr{H}(t_i)) = P_i^*(t_i, m_i)$ is parameterized by $\boldsymbol{h}_i$, so $P_i(t_i, m_i|\mathscr{H}(t_i)) = P_i(t_i, m_i|\boldsymbol{h}_i)$. P_i^* can be defined by f_i^*, F_i^*, S_i^* or ϕ_i^* (see Table 5.2) [270].

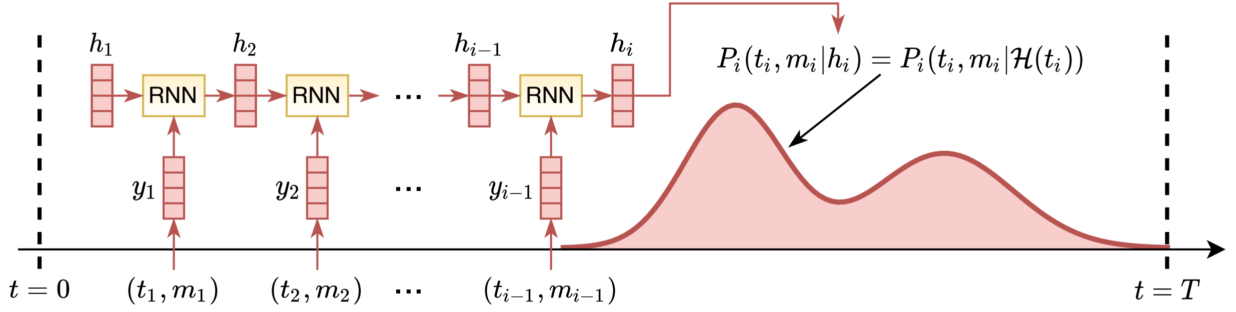


Figure 5.9.: In a neural TPP, the distribution over the next event $P_i(t_i, m_i|\mathscr{H}(t_i))$ is parameterized with the Recurrent Neural Network (RNN)'s hidden state vector $\boldsymbol{h}_i$, which encodes the event history $\mathscr{H}(t_i)$ [270].

While the first and second steps are similar for prominent neural TPP implementations such as *Recurrent Marked Temporal Point Processes (RMTPPs)* [93], *FullyNN* [213], and *LogNormMix* [269], they differ significantly in the third step. Therefore, in the following subsections, we present the neural TPP models *RMTPPs* [93] and *LogNormMix* [269] in more detail.

Recurrent Marked Temporal Point Processes (RMTPPs): The RMTPPs model was the first TPP to encode event history by the hidden state $\boldsymbol{h}_i$ of an RNN, thereby parameterizing the distribution over the next event $P_i^*(\tau_i, m_i)$, i.e., $P_i(\tau_i, m_i|\boldsymbol{h}_i)$. The model assumes conditional independence between the mark and inter-event time, such that $P_i(\tau_i, m_i|\boldsymbol{h}_i) = P_i(\tau_i|\boldsymbol{h}_i)P_i(m_i|\boldsymbol{h}_i)$. The mark distribution $P_i^*(m_i)$ is a categorical distribution. The time distribution $P_i^*(\tau_i)$ is characterized by the hazard function $\phi_i(\tau_i|\boldsymbol{h}_i) = \exp(w\tau_i + \boldsymbol{v}^T\boldsymbol{h}_i + b)$, where the vector $\boldsymbol{v}$ and the scalars b and w are learnable parameters and the exp transformation guarantees the positivity constraint of the hazard function [93]. By applying equation (5.7), we can express $\phi_i^*(\tau_i)$ as a conditional probability density function $f_i^*(\tau_i)$, which in this case is a Gompertz distribution [269]. Because of the simplicity of the hazard function, the integral $\int_0^{\tau_i} \phi_i^*(u)\, du$ of the likelihood can be computed analytically. Unfortunately, no closed-form formula exists for computing the mean of the distribution, i.e., $\mathbb{E}\left[f_i^*(\tau_i)\right]$. Instead, an integral must be solved numerically for its computation. However, the model allows for drawing samples analytically from the distribution [93].

LogNormMix: As with RMTPPs, the TPP LogNormMix [269] assumes conditional independence between the mark and time, such that $P_i(\tau_i, m_i|\boldsymbol{h}_i) = P_i(\tau_i|\boldsymbol{h}_i)P_i(m_i|\boldsymbol{h}_i)$. Similarly, the mark distribution $P_i^*(m_i)$ is defined as a categorical distribution. The unique feature of LogNormMix is that it characterizes the distribution over τ_i with the conditional probability density function $f_i^*(\tau_i)$, whereas most other TPP models use the intensity for this purpose. This offers the advantage that we can specify f_i^* with any positive probability density function, thereby automatically satisfying the condition of a valid distribution. LogNormMix uses a mixture model to specify f_i^*, as they are well suited for low-dimensional density estimations [191] and therefore in particular for modeling the one-dimensional inter-event time τ_i. As a mixture distribution defined in $(0, \infty)$, LogNormMix uses a mixture of K log-normal distributions defined by

$$f_i(\tau_i|\boldsymbol{w}_i, \boldsymbol{\mu}_i, \boldsymbol{s}_i) = \sum_{k=1}^{K} \frac{w_{ik}}{\tau_i s_{ik}\sqrt{2\pi}} \exp\left(-\frac{(\log\tau_i - \mu_{ik})^2}{2s_{ik}^2}\right) \tag{5.9}$$

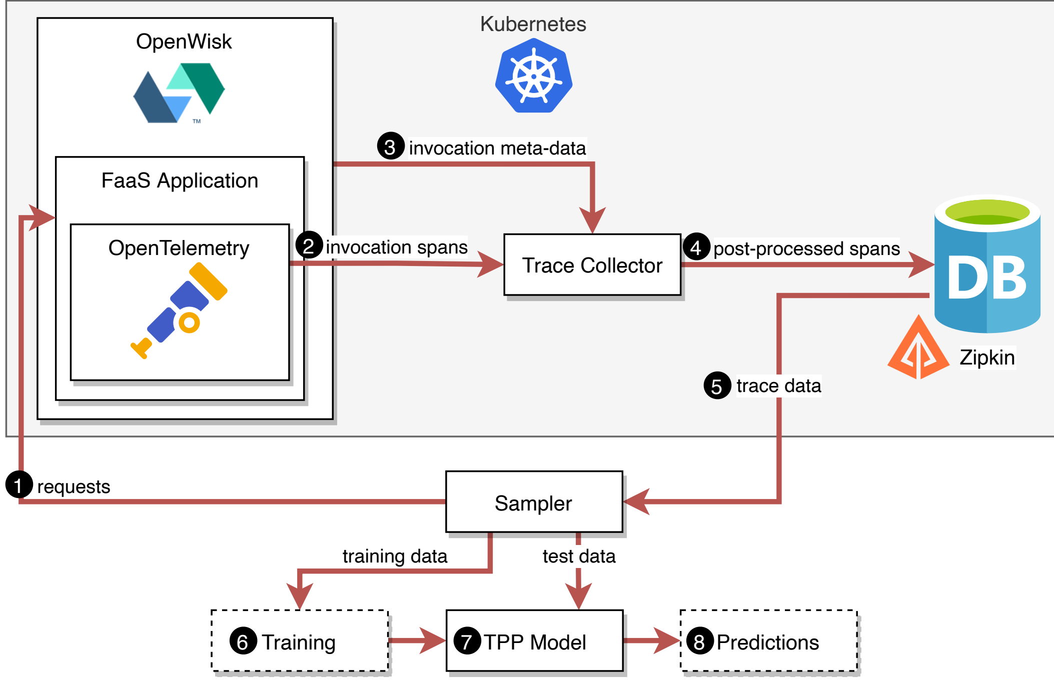


Figure 5.10.: *TppFaaS* is a system for modeling serverless applications using TPPs. For this purpose, trace data is collected from synthetic serverless applications that the user can easily create via configuration. The trace data is then used to train a TPP, which models the interactions between the functions in the application.

The parameters of the mixture distribution are computed using the hidden state $\boldsymbol{h}_i$ of the RNN, i.e.

$$\begin{aligned}
\boldsymbol{w}_i &= \mathrm{Softmax}(\boldsymbol{V}_w \boldsymbol{h}_i + \boldsymbol{b}_w) \\
\boldsymbol{s}_i &= \exp(\boldsymbol{V}_s \boldsymbol{h}_i + \boldsymbol{b}_s) \\
\boldsymbol{\mu}_i &= \boldsymbol{V}_\mu \boldsymbol{h}_i + \boldsymbol{b}_\mu
\end{aligned} \tag{5.10}$$

where $\boldsymbol{V}_w$, $\boldsymbol{b}_w$, $\boldsymbol{V}_s$, $\boldsymbol{b}_s$, $\boldsymbol{V}_\mu$, and $\boldsymbol{b}_\mu$ are learnable parameters and the softmax and exp transformations enforce the parameter constraints of the distribution. The model allows the computation of the survival function $S_i^*(T)$ of the likelihood with a closed-form formula. The mean of the distribution, i.e., $\mathbb{E}\left[f_i^*(\tau_i)\right]$, can also be computed analytically by taking the weighted mean of the component means. In addition, we can also analytically draw samples from the distribution [269].

We can efficiently train both models due to their likelihood in closed-form. However, the multimodal log-normal mixture distribution of LogNormMix provides much higher flexibility in modeling $f_i^*(\tau_i)$ than the unimodal Gompertz distribution of RMTPPs. Using a log-normal mixture distribution allows us the approximation of any distribution [269].

5.2.2 TppFaaS - Developed System

In this section, we give a brief overview of our developed system called *TppFaaS*, for building *Functions Interaction Model* using TPPs [279]. It currently supports OpenWhisk serverless compute platform. The process starts by deploying a serverless application on the OpenWhisk platform. The application's functions

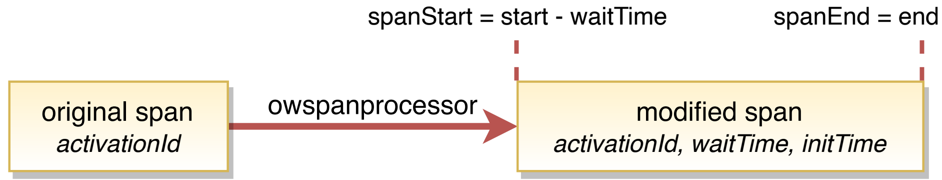


Figure 5.11.: The owspanprocessor adapts start and endpoint of the original span and adds further attributes.

are instrumented with *OpenTelemetry* library [222]. For generating the traces, user requests are sent to the deployed application (Step ❶ in Figure 5.10). The OpenWhisk executes the application. We created a custom *Trace Collector* that post-processes the spans produced by the instrumented application (Step ❷). The collector consists of one receiver, three processors, and one exporter. The spans produced by the instrumented application are received over HTTP by the pre-implemented **OLTP Receiver** [221] component of the *Trace Collector* and forwards to the first processor in the pipeline, the batch processor. The **Batch Processor** [220] aggregates the data to minimize later outgoing connections from the exporter. The next processor in the pipeline, the **owspanprocessor**, receives the aggregated spans. The processor extracts the span's *activationId* attribute to retrieve meta-information about the span's associated function invocation from the OpenWhisk API (Step ❸). The attributes extracted by the owspanprocessor measured in milliseconds are: *start*, *end*, *waitTime*, and *initTime*.

- The *start* attribute is a Unix timestamp and is computed by $start := executionStart - initTime$, where *executionStart* specifies the start time of the function code execution. That is, *start* already specifies the start of function initialization for a cold function invocation.
- The *end* attribute specifies the end of function execution.
- The *initTime* attribute specifies the duration of function initialization which applies only to cold function invocations, making the attribute optional.
- The *waitTime* attribute specifies the OpenWhisk caused delay occurring before the function initialization/execution [224].

The processor uses the extracted attributes to adjust the start and end time of the span as follows and is shown in Figure 5.11.

$$\begin{aligned} spanStart &:= start - waitTime \\ spanEnd &:= end \end{aligned} \tag{5.11}$$

Additionally, the *waitTime* and *initTime* are added to the modified span as attributes. The **owspanattacher** processor creates a child span for each of the *waitTime*, *initTime*, and *executionTime* that represents the function code execution as shown in Figure 5.12. The start time of the child span *executionTime* is computed with $executionStart = start + initTime$. The spans are then exported to *Zipkin* [320] (Step ❹). We use Cassandra as a backend database for Zipkin.

5.2.2.1 Sampler

The *Sampler* is an automated end-to-end pipeline containing all the necessary steps for trace datasets generation used to train and evaluate TPP models, such as deploying the application, sending requests, and

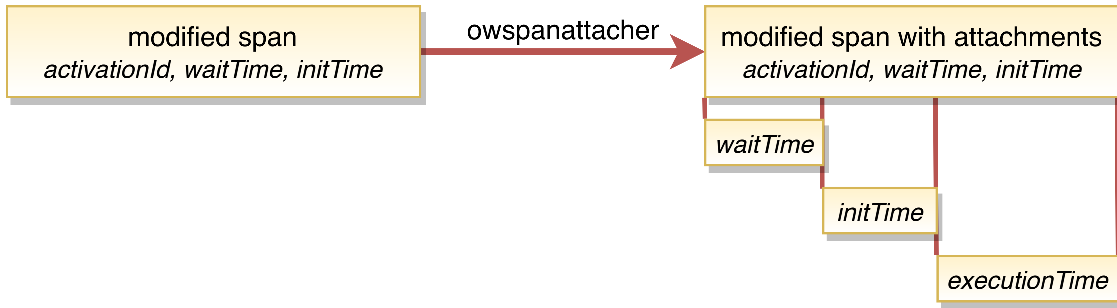


Figure 5.12.: The owspanattacher adds child spans for waitTime, initTime, and executionTime.

collecting data. The *Sampler* creates the datasets by sending `n` requests to the serverless application's `main` function at irregular time intervals. The `main` function represents the application's entry point. The time intervals between requests are drawn from a continuous uniform distribution with an interval specified by the user, who thus determines the load on OpenWhisk and, indirectly, the number of cold starts. Another feature of the *Sampler* is performing requests in batches, pausing requesting after each batch for a user-specified duration. The result is a dataset consisting of `n` traces whose format is compatible with training a TPP model.

The first step of the pipeline validates the user-input arguments, such as that the interval of the uniform distribution is in the positive range. Next, it verifies that Node.js and the Serverless framework [259] are available. Using Node.js, the pipeline installs the application's dependencies, such as the OpenTelemetry library. In the next step, the application is deployed using the Serverless framework, where the OpenWhisk credentials are read from a configuration file and provided to the framework as environment variables. After deploying the application, the *Sampler* sends `n` requests to the application's `main` function at irregular time intervals, whose durations are drawn from a uniform distribution each time. For each request, OpenWhisk returns the unique `activationId` of the `main` function invocation, which is collected in the `unfetched_ids` array. Once the *Sampler* has sent all `n` requests, it may take some time to execute all function invocations, depending on OpenWhisk's load. With the `activationIds` returned by OpenWhisk, we can reference any span associated with a `main` function invocation of the generated `n` traces. The Zipkin API provides the ability to filter traces by a single span attribute. Thus, iterating over the `activationIds` of the `unfetched_ids` array and setting the ID as a filter criterion, we fetch each trace of the `n` requests from the Zipkin API. For each fetched trace that is complete, the respective `activationId` is removed from the `unfetched_ids` array. If the trace is incomplete, we keep the ID in the array so that the trace can be retrieved again in the next loop. The iteration stops if either the array `unfetched_ids` is empty or the number of IDs in the array stagnates after several iterations. The latter happens upon runtime errors of OpenWhisk, so some traces are never completed. Zipkin returns the traces as JSON, from which the *Sampler* extracts the necessary information and converts it to a format compatible with the TPP model (Step ❺).

In order to convert the extracted spans into the TPP model format compatible, we first decompose the span of a function invocation into the three-time ranges: *waitTime*, *initTime* (for a cold start), and *executionTime*. We map the span to an instantaneous point in time, denoted as an event in the context of TPPs. For the next invocation, we want to predict when its request will arrive at the serverless compute platform. The platform could use the predicted time to upscale the function upfront, allowing it to begin its execution without delay. In reality, however, a cold start or platform-specific issues, such as the creation of a docker container, might delay the function execution, which OpenWhisk captures through the *waitTime* and *initTime*. So, to predict the time when the request for the next function invocation arrives at the platform, we need to subtract these

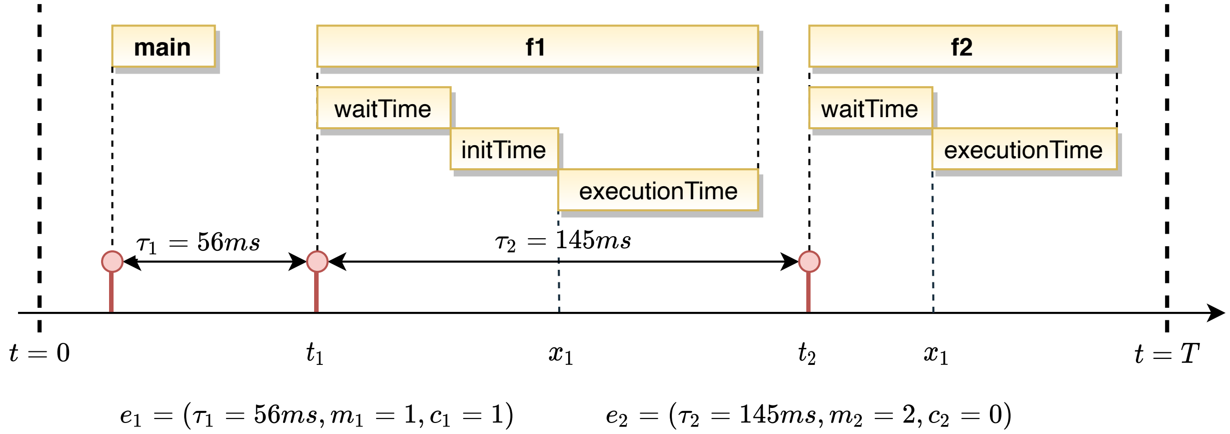


Figure 5.13.: The spans of the invoked functions *f1* and *f2* are mapped to the 3-tuple events e_1 and e_2, which carry the inter-event time τ_i, the function class m_i, and the cold start feature c_i. Given the cold invocation of *f1*, we have $c_1 = 1$.

delays from the actual start of the function. Let w_i be the *waitTime*, i_i be the *initTime*, and x_i be the start time of the function execution of the ith invocation, then we define $t_i = x_i - w_i - i_i$ as the mapping of the span to an instantaneous point in time. The mapping is visualized in the example in Figure 5.13, where the functions `main`, *f1*, and *f2* are invoked sequentially, with a cold start occurring on *f1*. We represent the event of the ith invocation, which we denote by e_i, as either the 2-tuple (τ_i, m_i) or 3-tuple (τ_i, m_i, c_i). The attribute $\tau_i = t_i - t_{i-1} \in \mathbb{R}_+$ describes the inter-event time. We use $m_i \in \mathbb{N}_0$, denoted as a mark, to specify the class of the invoked function. The binary attribute $c_i \in \{0, 1\}$ is an optional feature intended to enhance the predictive ability of the TPP model, indicating whether the ith function invocation was a cold start or not. We compute the feature with $c_i = w_i > 0$.

In the final steps of the pipeline, the *Sampler* saves the formatted trace dataset as a pickle. Once a trace dataset is generated, we split it into a training and test dataset. We use the training data to optimize the parameters of the TPP model (Steps ❻-❼), which we then evaluate using the test data (Step ❽).

5.2.2.2 TPP Models

In this section, we briefly describe the TPP models and their purposes within *TppFaaS*.

LogNormMix: τ_i as a Log-Normal mixture distribution: We use the TPP model LogNormMix (§5.2.1.1) to model the duration until the next function invocation with the conditional probability distribution $f_i^*(\tau_i)$, where f_i^* is defined as a log-normal mixture distribution. For this, we compute the duration until the next function invocation, i.e., the inter-event time τ_i, from the time points of the function invocations t_i. Since the inter-event times may take high values, we use their log values. The inter-event time is combined with the function class attribute m_i and the optional cold start feature c_i to yield the 3-tuple event $e_i = (\tau_i, m_i, c_i)$, which represents the function invocation and is input to the RNN. We represent each function class by a trainable 32-dimensional embedding vector. The vectors are concatenated into an embedding matrix indexed by m_i. We represent the two values of the cold start feature, c_i each, by a trainable 32-dimensional embedding vector. The RNN ingests the event e_i and produces a hidden state vector $h_i \in \mathbb{R}^{64}$ that encodes the history of past invocations. An affine transformation and subsequent softmax operation map the vector h_i to the parameters of the log-normal mixture distribution. The softmax operation forces the component weights of the mixture distribution to sum to 1.

TruncNorm: τ_i as a single value: Instead of an entire probability distribution $f_i^*(\tau_i)$, a single value for the inter-event time τ_i is sufficient for some applications. For example, if the serverless compute platform must initialize the function in advance to avoid a cold start, only the single value τ_i is required. Thus, we need a point estimate of $f_i^*(\tau_i)$ that maps the distribution to a single value. There are two methods to obtain this point estimate. First, we can model $f_i^*(\tau_i)$ with LogNormMix, which provides us with a log-normal mixture distribution for it. The expected value of this mixture distribution, i.e. $\mathbb{E}\left[f_i^*(\tau_i)\right]$, can be computed analytically and quickly, representing the desired point estimate of $f_i^*(\tau_i)$. In the second method, we map the hidden state vector h_i of the RNN to a positive real number representing the inter-event time τ_i using an affine transformation and subsequent softplus operation. Instead of softplus, we can use any other operation that enforces $\tau_i > 0$, such as the logarithm. We may also interpret this method as a TPP that models the conditional probability distribution $f_i^*(\tau_i)$ with a truncated normal distribution with constant variance [267]. The normal distribution is "truncated" as it is not defined in $\mathbb{R}$ as usual, but only in $\mathbb{R}_+$. The single value for τ_i, obtained by the affine transformation of h_i and the softplus operation, is the expected value of this distribution. In this work, we selected the second method (which we refer *TruncNorm*), since, for a simple point estimate, the high flexibility of the log-normal mixture distribution is unnecessary for modeling f_i^*. Moreover, we experienced more stable training with TruncNorm and a faster decrease of the loss function, i.e., the mean absolute error.

Mark Modeled with a categorical distribution: We assume that the mark or function type m_i and the inter-event time τ_i of the ith function invocation are independent. We define the distribution over m_i as the categorical distribution $f_i^*(m_i) = f_i(m_i|\mathscr{H}(t_i))$ parameterized by the vector $\boldsymbol{\pi}_i$. The value $\pi_{i,c}$ describes the probability that m_i is of class c. We obtain $f_i^*(m_i)$ by an affine transformation of the hidden state vector h_i produced by the RNN and a subsequent softmax operation.

5.2.3 Evaluation Settings

This section describes the various evaluation settings used to build the *Functions Interaction Model*. First, we describe the benchmark applications used in §5.2.3.1. Then we present the infrastructure settings on which the evaluation is conducted in §5.2.3.2. Furthermore, we explain the various datasets generated for evaluation in §5.2.3.3, and training models hyperparameters in §5.2.3.4. Lastly, in §5.2.3.5, we define the performance quality evaluation metrics used for evaluation of the results.

5.2.3.1 Benchmark Applications

To generate trace datasets, we construct several instrumented serverless applications. The applications are a composition of several artificial functions whose execution time is simulated by a sleep command. We configure the application in the YAML file of the Serverless framework. With it, we specify the call graph, i.e., the structure of the composition that dictates in which order the functions invoke each other. In addition, we use the configuration to specify the duration of the sleep commands of the individual functions. By adjusting these two hyperparameters, the structure of the composition, and the distribution of the function duration, we build applications with unique characteristics that complicate the modeling of the function invocations for the TPP model. In particular, 1) the constructed applications exhibit different structural characteristics (sequence, parallel, tree, and fanout), 2) each of the applications are scaled in two variants: small variant and large variant, and 3) for each variant of the application, we implement a randomized and a non-randomized variant. In the non-randomized variant, the duration of the sleep command for all functions is fixed with either `300ms`, `400ms` or `500ms`. The duration is drawn from a gamma distribution for each function invocation in the randomized variant. During configuration, we, therefore, assign each function one of three gamma distributions with expected values of either `300ms`, `400ms` or `500ms`.

5.2.3.2 Infrastructure Settings

Generating trace data with cold starts imposes high demands on the infrastructure. To meet these, we host the performance-critical components of the system architecture, i.e., OpenWhisk, the Trace Collector, and Zipkin, on Google Kubernetes Engine. Our Kubernetes cluster consists of nine nodes, each with 32 GiB of memory and a CPU (Intel Skylake architecture) with eight virtual cores. So, in total, we have 72 CPU virtual cores and 288GiB of memory at our disposal. The *Sampler* service requires only a few resources and runs on a separate VM with two virtual cores and 4 GiB of memory. We train our TPP models on a single-node cluster with 754 GiB of memory and two Intel Cascade Lake processors (Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz) with 22 cores each.

5.2.3.3 Dataset Generation

We generate datasets with 1000 traces each for all variants of the four applications sequence, parallel, tree, and fanout described in §5.2.3.1 [279]. The parallel, tree, and fanout applications each exist in a small and large variant, identified by the substring *small* and *large*, respectively, in the application name. In addition, each small and large variant and sequence exist in a randomized and non-randomized variant. We generate a dataset with and without cold starts for each of these variants. In the former, the cold invocations account for exactly 30% of the total invocations. To generate such a dataset, we create 400 traces with almost exclusively cold invocations and 1000 traces with almost exclusively warm invocations. We then incrementally substitute the warm traces with cold traces until the 30% of cold invocations is reached. We generate the datasets using the *Sampler*, which sends requests to a given application. The duration between requests is drawn each time from a continuous uniform distribution whose interval bounds are specified by the parameters l (lower bound) and u (upper bound). Thus, the interval specification influences the request rate and, thus, the load on OpenWhisk. A higher request rate increases the load on OpenWhisk, which responds by scaling up the functions, causing cold starts. These interval limits are accordingly set to generate datasets with or without cold starts.

5.2.3.4 Training Details and Model Parameters

We partition the 1000 traces of each dataset into 600 for training and 200 each for validating and testing the TPP model. The training set is used to optimize the model parameters, the validation set is used for evaluation during training, and the test set is used for the final evaluation. We train and evaluate each dataset to obtain averaged results using ten different splits. For each split, we train two TPP models, LogNormMix and TruncNorm. We optimize the former with the loss function $\mathscr{L}_{\text{NLL}}$ and the latter with $\mathscr{L}_{\text{MAE}}$. Both loss functions evaluate the prediction of the next function class m_i with the Negative Log-Likelihood (NLL), but differ in the evaluation of the predicted τ_i. LogNormMix predicts τ_i with the conditional probability distribution $f_i^*(\tau_i)$, whereas TruncNorm provides a concrete value for τ_i, which we denote with τ_i^{pred}. The loss function $\mathscr{L}_{\text{NLL}}$ evaluates the distribution $f_i^*(\tau_i)$ using the NLL, whereas the loss function $\mathscr{L}_{\text{MAE}}$ computes the Mean Absolute Error (MAE) for τ_i^{pred}. To derive $\mathscr{L}_{\text{NLL}}$, we denote by $x = \{e_1 = (\tau_1, m_1), \dots, e_N = (\tau_N, m_N)\}$ an event sequence representing a trace of invocations. The likelihood of the trace is defined by

$$p(x|\theta) = \prod_{i=1}^{N} \left[f_i^*(\tau_i, m_i) \right] S_{N+1}^* \tag{5.12}$$

Assuming that the inter-event time τ_i and mark m_i are independent, we obtain our loss function:

$$\begin{aligned} p(x|\theta) &= \prod_{i=1}^{N} \left[f_i^*(\tau_i, m_i) \right] S_{N+1}^* \\ &= \prod_{i=1}^{N} \left[f_i^*(\tau_i) f_i^*(m_i) \right] S_{N+1}^* \\ \mathscr{L}_{\text{NLL}}(\theta) &= -\log p(x|\theta) \\ &= -\sum_{i=1}^{N} \left[\log f_i^*(\tau_i) + \log f_i^*(m_i) \right] - \log S_{N+1}^* \end{aligned} \tag{5.13}$$

The model parameters are optimized by minimizing the loss function. For this, we use the optimization algorithm Adam [157] with a learning rate of 10^{-3} and minibatch size of `64`. We train LogNormMix and TruncNorm up to `2000` and `4000` epochs, respectively, where an epoch describes the iteration over the entire training data. If the loss does not decrease after `100` and `200` epochs, respectively, for the validation set, we abort the training and pick the model with the lowest loss for the validation set. To reduce the effect of overfitting, we apply L2 regularization with 10^{-5} on the model parameters. To model $f_i^*(\tau_i)$, LogNormMix uses a log-normal mixture distribution with $K = 64$ components. According to [269], the parameter K does not impact the model's performance, which is why we do not test any other values. As RNN architecture, we use a Gated Recurrent Unit (GRU) [77] with a hidden state vector in $\mathbb{R}^{64}$.

5.2.3.5 Performance Quality Measures

The TPP **LogNormMix** predicts the conditional probability distribution $f_i^*(\tau_i)$ over the inter-event time τ_i and the conditional categorical distribution $f_i^*(m_i)$ over the marks m_i. We use the NLL to evaluate the predicted distributions with respect to the test dataset $x = \{(\tau_1, m_1), \ldots, (\tau_N, m_N)\}$. Using NLL_{time}, NLL_{mark}, and $\text{NLL}_{\text{total}}$, we evaluate the distribution over τ_i, m_i, and both variables, respectively. The NLL quality measures are defined as follows:

$$\begin{aligned} \text{NLL}_{\text{time}} &= -\frac{1}{N} \sum_{i=1}^{N} \log f_i^*(\tau_i) - \log S_{N+1}^* \\ \text{NLL}_{\text{mark}} &= -\frac{1}{N} \sum_{i=1}^{N} \log f_i^*(m_i) \\ \text{NLL}_{\text{total}} &= \text{NLL}_{\text{time}} + \text{NLL}_{\text{mark}} \end{aligned} \tag{5.14}$$

It is worth noting here that a single NLL value has little explanatory power. That is, we cannot evaluate whether a value is "good" without referring to other values. For this reason, the relative differences between the NLL values for different datasets is analyzed [269].

The *accuracy* is another quality measure that evaluates LogNormMix's predictive capability of the mark m_i. It describes the fraction of correctly predicted marks, such that `1.0` is the optimal and `0.0` is the worst value for this metric. We obtain the predicted class c^{pred} of the mark m_i with

$$c^{\text{pred}} = \arg\max_{c} \pi_{i,c}, \tag{5.15}$$

where $\pi_{i,c}$ describes the probability that the ith function invocation is of class c. We expect a correlation between the measure NLL_{mark} and the accuracy. The accuracy measure evaluates the TPP according to its

capability to predict a single class for the next function invocation. The serverless compute platform can use the prediction to scale the corresponding class in advance.

The TPP **TruncNorm** predicts a single value for the inter-event time τ_i and also, like LogNormMix, a conditional categorical distribution over m_i. We evaluate the predicted value for the inter-event time, denoted as τ_i^{pred}, by computing the MAE for the test dataset. Besides the mean value of the absolute errors, the distribution of the errors is interesting. This gives us information if the time predicted for the invocation was too early or too late. Like LogNormMix, TruncNorm also predicts a distribution over the mark m_i. However, in contrast to LogNormMix, we do not evaluate this distribution because the results of the two TPPs would be similar. This is because both predict their mark distribution conditionally independent of the time. Therefore, the distribution is only conditioned on the history embedding h_i produced by an RNN in both TPPs.

5.2.4 Results

We evaluate our TPP models LogNormMix and TruncNorm with respect to various applications (§5.2.3.1), which differ in structure, number of functions, and randomization of the function's sleep command. In this subsection, we present the results of both datasets (with and without cold starts). We evaluate the predicted distributions with the NLL and predicted single values with the MAE. For both quality measures, lower values are better, and zero is optimal. A single NLL value has little explanatory power. Instead, the differences between values for different applications are of interest. In contrast, a single MAE value is meaningful and valuable even without comparison to other values.

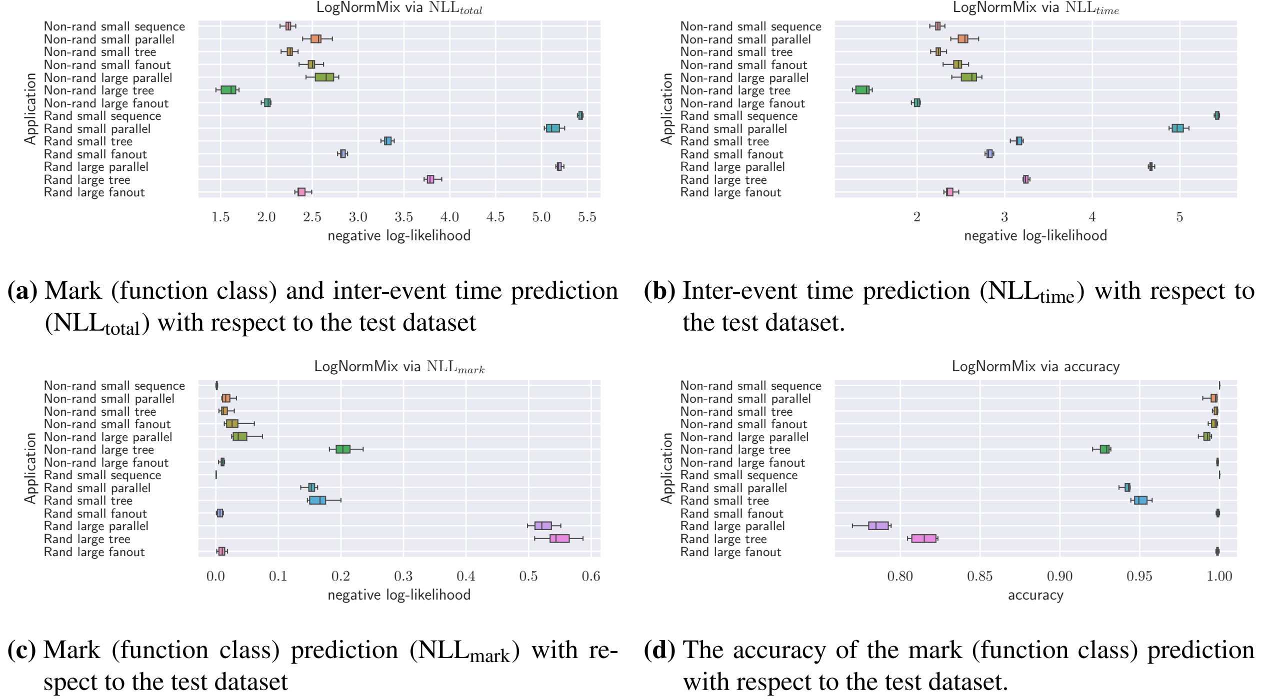
5.2.4.1 Predictions on Datasets without Cold Starts

In this section, we present the results of prediction on dataset without cold starts.

LogNormMix via $\text{NLL}_{\text{total}}$: LogNormMix predicts a distribution for the inter-event time τ_i and for the mark m_i, i.e., for the function class. Using $\text{NLL}_{\text{total}}$ from equation (5.14), we evaluate both distributions combined and present the results in Figure 5.14a. Looking at the NLL measures: $\text{NLL}_{\text{total}}$, NLL_{time}, and NLL_{mark} in Figure 5.14a, Figure 5.14b and Figure 5.14c, we notice that NLL_{time} has a much higher proportion in $\text{NLL}_{\text{total}}$ than NLL_{mark}. For example, the application tree_large_rand has a value of about `3.8` for $\text{NLL}_{\text{total}}$. In this value, about `3.25` accounts for NLL_{time} and about `0.55` for NLL_{mark}. Thus, we can infer that it is much more challenging for LogNormMix to predict the time than the functional class. Future research should therefore prioritize improving the prediction of the inter-event time τ_i.

LogNormMix via NLL_{time}: We evaluated the inter-event time with NLL_{time} from equation (5.14) and show the results in Figure 5.14b. We draw the following conclusions:

- **Differences between Randomized and Non-Randomized Applications**: A look at the metric NLL_{time} in Figure 5.14b shows that LogNormMix performs better for non-randomized applications than for randomized ones. This is expected since, for random, the function duration is drawn from a gamma distribution instead of being constant. Distributions of the inter-event time τ_i for randomized applications have a higher variance than for non-randomized ones. This higher variance makes prediction more challenging for the TPPs. The fact that the function duration is drawn independently of the gamma distribution also impairs the prediction. Some dependency between function execution times can be assumed in a real-world application. For example, suppose the execution time of a function is longer than usual due to a high load on the serverless compute platform. In that case, the following



(a) Mark (function class) and inter-event time prediction ($\mathrm{NLL}_{\mathrm{total}}$) with respect to the test dataset

(b) Inter-event time prediction ($\mathrm{NLL}_{\mathrm{time}}$) with respect to the test dataset.

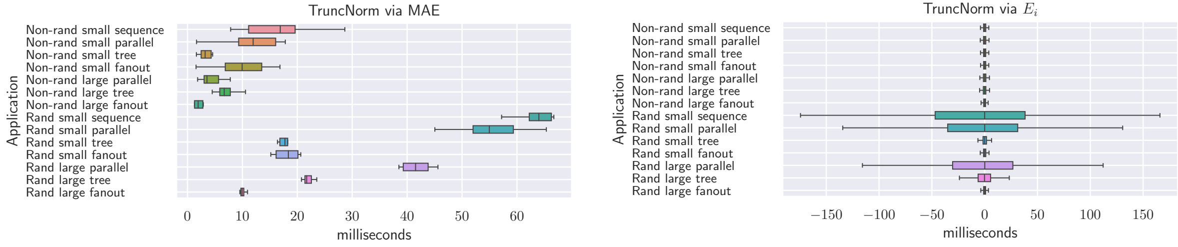
(c) Mark (function class) prediction ($\mathrm{NLL}_{\mathrm{mark}}$) with respect to the test dataset

(d) The accuracy of the mark (function class) prediction with respect to the test dataset.

Figure 5.14.: LogNormMix evaluated via the NLL, with the datasets having no cold starts. A lower value is better, and zero is optimal, except for accuracy, where a higher value is better, and `1.0` is optimal.

functions will likely execute longer than usual. The information about the overload will be encoded in the higher inter-event times, thus improving the prediction time of the TPP. Furthermore, we see in Figure 5.14b that the results for applications with a small proportion of parallel functions suffer particularly from randomization. For example, this is evident for the applications sequence_small, which has no parallel functions, and fanout_large, which has a high proportion of parallel functions. While the TPP performs marginally worse in the non-randomized case for sequence_small than for fanout_large (difference of approximately `0.25`), this difference is much more significant in the randomized case (difference of approximately `3`). This is because each function has a successor invoked after a sleep command completes in sequence. This means that there is a randomized sleep command between every two invocations, making predictions more difficult. In contrast, the parallel functions in fanout are invoked as a sequence without any intermediate randomized sleep commands, so the results in fanout are less affected by the randomization.

- **Differences between Small and Large Applications**: We observe from Figure 5.14b that the result for the applications parallel_small and parallel_large are equal in the non-randomized case, but the result for parallel_large is slightly better in the randomized case. It contradicts our assumption that a higher number of parallel function branches will affect the prediction performance for the inter-event time. Moreover, in Figure 5.14b LogNormMix performs better for tree_large than for tree_small in the non-randomized case, and the results of both applications are equal in the randomized case. It indicates that a higher tree depth does not negatively influence prediction performance. In addition, we see in Figure 5.14b that the prediction performance for fanout_large is better for fanout_small, which is due to the higher proportion of parallel functions. It also shows that scaling the number of parallel functions in the application structure does not harm the prediction time of the TPP.



(a) Via the Mean Absolute Error (MAE) of the inter-event time prediction.

(b) Via the distribution of the errors between the predicted and true inter-event time ($E_i = \tau_i^{\text{pred}} - \tau_i$).

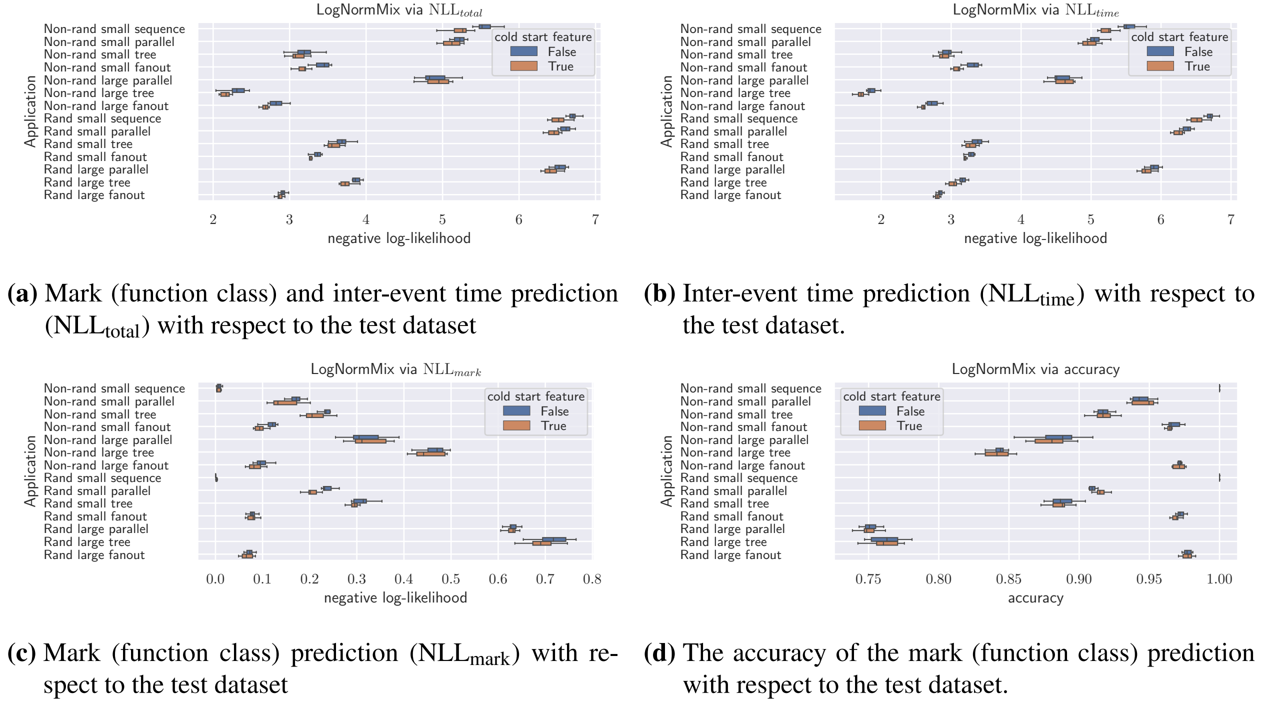
Figure 5.15.: TruncNorm evaluation on dataset with no cold starts. The lower value is better, and zero is optimal. A negative value indicates that the predicted time for the invocation was too early.

LogNormMix via NLL$_{\text{mark}}$: In addition, we evaluate the function class distributions separately with NLL$_{\text{mark}}$ from equation (5.14) and show the results in Figure 5.14c. The NLL$_{\text{mark}}$ measure in Figure 5.14c shows that LogNormMix performs well for the majority of the applications, i.e., the values are close to zero. However, exceptions are the results for tree_large and the randomized versions of parallel and tree. A drop in performance between the small and the large versions can be observed for the two latter applications, parallel and tree. The characteristic of these applications' structure is a high number of parallel function branches. It indicates that the function class prediction is challenging for applications with this structure. Since the function class order is the same for all traces, LogNormMix performs best for the application sequence with a near-zero NLL value.

LogNormMix via Accuracy: Another measure that evaluates the performance of the function class prediction is accuracy. The measure is defined in the range $[0.0, 1.0]$, where `1.0` is the best (all classes were predicted correctly), and `0.0` is the worst. We show the results of LogNormMix for this measure in Figure 5.14d. The results of the accuracy in Figure 5.14d reflect the results of the NLL$_{\text{mark}}$ measure, though the values are more interpretable. We see that LogNormMix achieves accuracy close to `1.0` for most applications, meaning that almost all invocations are classified correctly. Analogous to NLL$_{\text{mark}}$, LogNormMix achieves worse results for the randomized versions of parallel and tree. However, an accuracy of above `0.93` is still achieved for tree_large, parallel_small_rand, and tree_small_rand, which is acceptable. On the other hand, an accuracy of about `0.8` for parallel_large_rand and tree_large_rand could further be improved by collecting more data.

TruncNorm via MAE: TruncNorm predicts a single value for the inter-event time τ_i. We evaluate this prediction using the MAE and show the results in Figure 5.15a. Figure 5.15a shows the results of TruncNorm's inter-event time predictions in terms of the MAE. The results are similar to those for the NLL$_{\text{time}}$ measure, i.e., they exhibit the same patterns: better results for non-randomized applications than for randomized ones, smaller drop in performance due to randomization for applications with a higher proportion of parallel functions (e.g., tree and fanout), and no negative impact on the results when scaling the application structure from small to large. For the non-randomized applications, all MAE values are below `20ms`, which is reasonable given the duration of the sleep command from `300ms` to `500ms`. It also applies to the randomized applications, excluding the applications parallel and sequence. For these two applications, the values of about `40ms` and `60ms` can be improved by providing more features for the TPP in future work.

TruncNorm via E_i: In addition, we calculate the errors E_i for the entire test dataset and visualize their distribution in Figure 5.15b. Here, lower absolute values are better, and zero is optimal. The error distributions of the inter-event time predictions in Figure 5.15b show that TruncNorm performs well for most applications. However, analogous to the results for the MAE, the performance for the randomized versions



(a) Mark (function class) and inter-event time prediction ($\mathrm{NLL}_{\mathrm{total}}$) with respect to the test dataset

(b) Inter-event time prediction ($\mathrm{NLL}_{\mathrm{time}}$) with respect to the test dataset.

(c) Mark (function class) prediction ($\mathrm{NLL}_{\mathrm{mark}}$) with respect to the test dataset

(d) The accuracy of the mark (function class) prediction with respect to the test dataset.

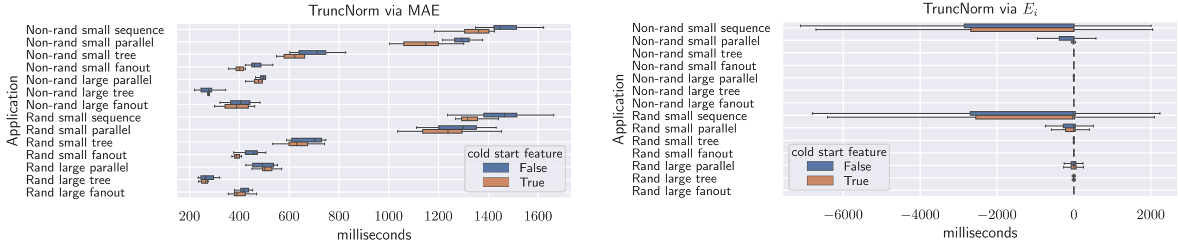
Figure 5.16.: LogNormMix evaluated via the NLL, with the datasets having `30%` of the invocations as cold starts. For each application, the TPP is trained and evaluated once with the cold start feature c_i enabled and once with it disabled. A lower value is better, and zero is optimal, except for accuracy, where a higher value is better, and `1.0` is optimal.

of sequence and parallel was relatively poor. Here, the error distributions have higher variances than for the other applications. Notably, all distributions are symmetric and centered to zero.

5.2.4.2 Prediction on Datasets with Cold Starts

This section repeats the evaluation from §5.2.4.1 but with the difference that `30%` of the function invocations are cold starts. Another difference is that we train and evaluate the models twice for each application. Once the cold start feature $c_i \in \{0,1\}$ is included in the event representation, i.e. (τ_i, m_i, c_i), and once it is not, i.e. (τ_i, m_i). The feature indicates whether the ith invocation is a cold-start.

LogNormMix via $\mathrm{NLL}_{\mathrm{time}}$: We evaluate the inter-event time with $\mathrm{NLL}_{\mathrm{time}}$ from equation (5.14) and present the results in Figure 5.16b. The results regarding $\mathrm{NLL}_{\mathrm{time}}$ in Figure 5.16b are similar to the results for this measure without cold starts in Figure 5.14b, yet with slightly poorer performance. However, one difference is that LogNormMix also performed relatively poorly for the non-randomized versions of the sequence and parallel applications. At the same time, this is not the case for the datasets without cold starts. The inter-event time distributions in the cold start datasets have a high variance for the sequence, parallel_small, and parallel_large applications. It affects the prediction performance. The inter-event time distributions in the datasets without cold starts have almost no variance. The high variance of the inter-event time distributions is caused by the high variance of the waitTime distributions. The high waitTime values, up to `10` seconds, are caused by the high load imposed on OpenWhisk to enforce cold starts. Furthermore, it can be seen in Figure 5.16b that the enabled cold start feature slightly improved the prediction results. However, the improvement is marginal as the significant uncertainty in inter-event time prediction comes from the waitTime



(a) Via the Mean Absolute Error (MAE) of the inter-event time prediction.

(b) Via the distribution of the errors between the predicted and true inter-event time ($E_i = \tau_i^{\text{pred}} - \tau_i$).

Figure 5.17.: TruncNorm evaluation for the dataset having `30%` of the invocations as cold starts. For each application, the TPP is trained and evaluated once with the cold start feature c_i enabled and once with it disabled. The lower value is better, and zero is optimal. A negative value indicates that the predicted time for the invocation was too early.

values with high variance. The feature provides the information that a cold start occurred and that a higher inter-event time can be expected. However, the prediction is still challenging due to the high variance of the waitTime.

LogNormMix via NLL$_{\text{mark}}$: Looking at the results of the NLL$_{\text{mark}}$ measure in Figure 5.16c, it is noticeable that they are slightly worse than the results for the datasets without cold starts in Figure 5.14c. It implies that the function class prediction was also affected by the higher variance of the inter-event time caused by the higher variance of the waitTime. Similar to the results without cold starts, LogNormMix performs worse for parallel and tree applications due to their structure with parallel function branches. Enabling the cold start features led to improvements, but as with the results for the NLL$_{\text{time}}$ measure, these are marginal.

LogNormMix via Accuracy: Similar to the drop in performance for NLL$_{\text{mark}}$ due to cold starts, the results with the accuracy measure in Figure 5.16d are also dropped. Especially, the results for the non-randomized versions of parallel and tree are affected by the high variance of waitTime. For example, the results for parallel_small and tree_small decrease by approximately `0.06` and `0.08`, respectively, compared to the results for the datasets without cold starts in Figure 5.14d. The highest decrease in accuracy of approximately `0.11` is experienced for the parallel_large application. Even though the performance is generally decreased due to the cold starts, the results are still good. The accuracy for parallel_large_rand and tree_large_rand decrease only by approximately `0.03` and `0.05`, respectively. Similarly, for parallel_small_rand and tree_small_rand decrease roughly by `0.03` and `0.06`, respectively. The accuracy for all versions of fanout decrease at most by `0.03`.

TruncNorm via MAE: Similar to the decrease in performance with respect to NLL$_{\text{time}}$ due to the cold starts, a decrease in performance with respect to the MAE in Figure 5.15a is also observed. The high variance of the waitTime in the cold start datasets significantly affects the prediction performance of TruncNorm, resulting in MAEs of more than `400ms`. Compared to the results for the datasets without cold starts in Figure 5.15a, where the MAE is below `20ms` for most applications, this is a significant increase. The MAE is especially high for the sequence and the small versions of parallel applications, with values between `1s` and `1.5s`. It could be related to the fact that the structures of these applications have a low proportion of parallel functions. In contrast, the performance of TruncNorm is relatively good for the large versions of the parallel application. It is surprising since LogNormMix struggled to predict the time for these applications, as seen in Figure 5.16b. We can also observe that the cold start feature improves the prediction performance, especially for the small versions of sequence and parallel applications.

TruncNorm via E_i: The error distributions of the inter-event time predictions in Figure 5.17b show that TruncNorm achieves good results for most applications, i.e., absolute values close to zero. Analogous to the results with MAE in Figure 5.17a, the performance for the small versions of parallel and especially sequence applications is relatively poor as the error distributions have high variance. Furthermore, the error distributions show that most of the errors were negative. By the definition $E_i = \tau_i^{\text{pred}} - \tau_i$, a negative error signifies that the predicted time for the invocation was too early. This is because the high waitTime delayed the invocation.

5.3 Summary

In this chapter, we presented two behavioral models based on the monitoring data collected by *FDN-Monitor* for characterization of the FaaS functions: 1) Functions Performance Model (§5.1), and 2) Function Interaction Model (§5.2).

In §5.1, we demonstrated the impact of various configuration parameters on the Function Capacity (FC) for the two serverless compute platforms (AWS Lambda and GCF). The introduced methodology and the tool *FnCapacitor* aim to solve the problem of estimating the FC at a certain deployment configuration. *FnCapacitor* can be used by application developers in an offline manner for estimating the FCs of FaaS functions within their application for different deployment configurations. The developer can deploy the functions with the right configurations based on the estimated FCs and the requirements. *FnCapacitor* can also be used by application developers in an online manner, where the tool collects the monitoring data of the already existing FaaS functions and builds the models automatically in the background without additional load testing. The built models can then be used to update the deployment configurations of the functions depending on the required SLOs. Creating a function scheduling based on the estimated FCs is a prospective future direction.

In §5.2 we have shown that neural TPPs effectively model the time and class of function invocations in a serverless application. For this purpose, we introduced *TppFaaS*, a system for creating synthetic serverless applications and using their collected data to train and test neural TPPs. In this data, function invocations are represented by the timing of their function trigger events. In addition, the data contains meta-information, such as the function class and the cold start initialization time. With these datasets, we trained and tested the two TPPs: LogNormMix and TruncNorm. It was shown that both models managed to capture the latent temporal dynamics of the different applications. We observed that the performance of the time prediction was not affected by scaling the application structure. Moreover, the function class prediction proved more challenging for applications involving parallel executed function branches. The TPPs performed well for all measures for datasets without cold starts. Here, LogNormMix achieved an accuracy of over `0.94` for most applications. Also, the MAE of TruncNorm's time prediction was below `22ms` for most applications. However, the predictions for the datasets with cold starts were more challenging. Here, TruncNorm achieved a MAE between `200ms` and `750ms` for most applications. The high errors resulted from the high variance of the waitTime, which measures the time an invocation request waits for execution in the internal OpenWhisk system. In addition, LogNormMix's function class prediction performance declined for the cold start datasets. Nevertheless, an accuracy above `0.85` was achieved for most applications, which is still satisfactory. The cold start feature, which indicates whether a cold start occurred, improved the results only marginally. Since the most uncertainty in the prediction is caused by the high variance of the waitTime and not by the variance of the cold start initialization time. Future work may provide additional features such as the number of invoker resources or the number of invocation requests waiting in the OpenWhisk system to the TPP to assist in estimating the estimation of the waitTime. In general, predicting the time is more complicated than predicting the functional class for datasets with and without cold starts.

6

Courier: Users's Functions Invocations Delivering and Load Balancing in FDN

"I have not failed. I've just found 10,000 ways that won't work."

— Thomas A. Edison

This chapter presents the Courier component of the FDN, responsible for delivering and load balancing user's functions invocations across the edge-cloud continuum in FDN. We start with the introduction to the Courier component in §6.1, explaining the challenges it needs to tackle and its two components: *Courier Load Balancer* and *Courier Control Plane*. We describe the design of the *Courier Load Balancer* in §6.2. We then present the *Courier Control Plane* in §6.3 along with the function delivery policies in §6.3.1 to select a subset of clusters. Lastly, in §6.3.2, we describe two load balancing algorithms that balance the users' invocations across the selected subset of clusters based on the function delivery policy.

6.1 Introduction

In order to deliver the incoming invocations across the target serverless compute clusters spread across the continuum, we need to create a load balancer. The load balancer receives the user's invocations and decides on a subset of serverless compute clusters to handle the invocations based on function-awareness and data-awareness. The invocations are load balanced across the selected subset of clusters based on the set load balancing algorithm. However, to do so, we need to tackle the following challenges:

- **Clusters awareness**: The load balancer must know which clusters are part of the FDN and their deployment location (at the edge, in the cloud, or on-premise) to route the requests to the appropriate clusters. Also, the load balancer needs to periodically perform health checks of the clusters to route the requests only to the healthy clusters and update the delivery policies accordingly.
- **Functions awareness**: Each function may not be deployed on all the clusters. Therefore, the load balancer must know which functions are deployed on which clusters. It will enable the load balancer to direct user requests for the functions to those clusters.

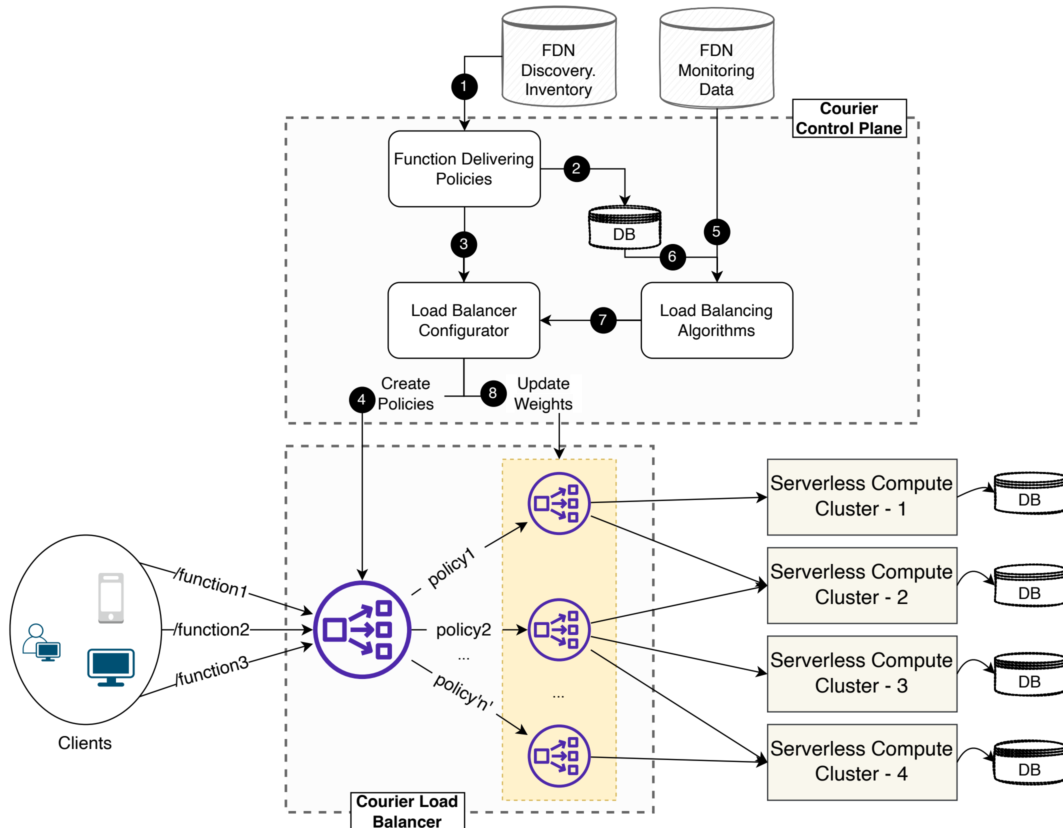


Figure 6.1.: A high-level overview design of the Courier component of the FDN, responsible for delivering and load balancing user's functions invocations across the edge-cloud continuum in FDN. Courier mainly consists of two components: *Courier Load Balancer* and *Courier Control Plane*. The *Courier Load Balancer* itself consists of two layers. The *Courier Control Plane* is responsible for configuring the *Courier Load Balancer*.

- **Data awareness**: Each function within an application has different data requirements, and all the data may not be available on all clusters due to privacy and resource constraints. Therefore, the load balancer must track the data distributed across the clusters and the functions' data requirements.

To this end, we create a tool called **Courier** within FDN responsible for delivering the invocations of the function to the suitable set of clusters based on function-awareness and data-awareness. The invocations are then load balanced across the selected subset of clusters based on the set load balancing algorithm. A high-level design of the **Courier** is shown in Figure 6.1. **Courier** mainly consists of two components: *Courier Load Balancer* and *Courier Control Plane*. *Courier Load Balancer*, as the name suggests, is the main entry point for the users and is responsible for load balancing users' functions invocations across multiple serverless compute clusters spread across the edge-cloud continuum based on the set configuration. The *Courier Load Balancer* itself consists of two layers. The first is the access point from the outside to the FDN, offering an HTTP/HTTPS endpoint. Successively, the requests are dispatched to the second layer using set function delivery policies (function-awareness and data-awareness), where a different load balancer is employed depending on the policy. The second layer of the *Courier Load Balancer* balances the users' invocations across the selected subset of clusters based on a set load balancing algorithm. On the other hand, the *Courier Control Plane* is responsible for configuring the *Courier Load Balancer* based on various function delivery policies and load balancing algorithms.

6.2 Courier Load Balancer

A central software system that can orchestrate users' functions invocations across multiple serverless compute clusters spread across the edge-cloud continuum is called *load balancer*. Load balancing happens most commonly during name resolution (via DNS) or the HTTP request [181, 209]. HTTP is the most commonly used protocol in today's internet landscape. It allows end-users to access websites via their browsers and builds the foundation for the REST paradigm - one of the most used paradigms to create APIs for web services. Therefore, *Courier Load Balancer* currently only considers function invocations made through HTTP endpoint triggers that specify the functions to be executed and possible parameters. These requests are simple GET or POST requests to the endpoint of the load balancer. The URL called is of the form:

```
http://{ADDRESS}:{PORT}/function/{FUNCTION_NAME}
```

The address and the port depend on the network instance of the load balancer, while the function name refers to one of the deployed functions. When *Courier Load Balancer* receives the invocation, it looks for the `X-FDN-Bucket` header. It uses its value and the set function delivery policy (§6.3.1) to select the appropriate subset of target serverless compute clusters and load balances among them based on the set load balancing algorithm. Load balancing in HTTP can generally be done in two different ways:

- via a *Reverse Proxy*: A user sends a request to an HTTP server, decides which server should handle it, and forwards it to this server. After the computation, the server returns the result to the reverse proxy, which returns the result to the user. In this method, every request passes through the reverse proxy. NGINX [208] and HAProxy [125] are some of the most known representatives of a reverse proxy.
- via *HTTP Redirects*: A user sends a request to an HTTP server, which decides which server should handle the request and redirects the user via a status code `3XX` to this server. The user will then connect directly to the server, which handles future requests.

Both methods offer advantages and disadvantages. The *Reverse Proxy* method offers the benefit of transparent load balancing to the user. Therefore, the user does not need to create another request (which is necessary with the redirect method). Another benefit of *Reverse Proxy* is that it can terminate the Secure Sockets Layer (SSL) session. Terminating SSL sessions simplify certificate management and leave encryption and decryption of computations to the *Reverse Proxy*. We, therefore, decided on load balancing via *Reverse Proxy* method.

Courier Load Balancer is based on HAProxy (High Availability Proxy), an open-source load balancer that offers a very reliable solution for TCP and HTTP-based applications [125]. HAProxy was initially released in 2001 and has become the de-facto standard open-source source load balancer used by many essential web enterprises such as Airbnb and GitHub [126]. Once the HAProxy has been deployed, it is only required to modify the configuration file that defines how the load balancer will work to control its behavior. We chose HAProxy due to its simplicity and the fact that it offers an interface for updating its parameters dynamically. HAProxy offers a runtime API [94] that allows modifying some settings during the program's execution without being forced to stop the whole system and, therefore, no downtime.

FDN consists of heterogeneous clusters spread across the edge-cloud continuum, and heterogeneous functions run within FDN have different data and compute requirements. For instance, functions heavily based on video processing can operate more efficiently if we co-locate executing functions and data physically. This is often best achieved by shipping code to data (in our case, executing the function on the clusters that have the data required by the function) rather than the approach of pulling data to code [127]. Therefore, the *Courier Load Balancer* is composed of two layers. The first is the access point from the outside to the

architecture, offering an HTTP endpoint. Successively, the function requests are delivered to the second layer using various function delivery policies (§6.3.1). Each function delivery policy selects a subset of the available clusters and employs a load balancer for load balancing the incoming requests to the selected backend clusters. *We mainly employ the Weighted Round-Robin (WRR) policy in the second layer of the load balancer for load balancing the incoming requests.* The weights are updated automatically based on different load balancing algorithms (§6.3.2 by the **Courier Control Plane**.

6.2.1 Courier Load Balancer Configuration

HAProxy allows building multiple load balancers using a single instance of it by its backend feature [125]. This way, we can run the entire framework on the same machine without adding extra network overhead due to the presence of two different layers. An example configuration file used to configure *Courier Load Balancer* is shown in Listing 6.1 based on the HAProxy configuration file [287]. The `frontend` section (line 1 in Listing 6.1) corresponds to the first layer. It defines how the user can connect to the load balancer, specifying the address and port on which it is listening. It is possible to create more interfaces by declaring different frontend sections, each of which must be connected to a different endpoint. The `http_front_courier` is simply the name we have given to it. `bind` (line 2 in listing 6.1) specifies the network interface on which the load balancer is listening. We use port 80 so that all the functions can be invoked using a normal HTTP connection. `stats` (line 3 in listing 6.1) specifies the URL used for accessing the metrics of the load balancer and is useful for analyzing and monitoring the behavior of the load balancer. It is possible to specify forwarding rules that are called Access Control Lists (ACLs) [136] (referred to as `acl` in lines 5-6, 8-9 in listing 6.1). They allow setting custom rules to block malicious requests, redirecting, or specify the target backend based on some policies. For example, an ACL can read the URL of an incoming request, analyze it, and forward it to the exact server according to its format. We use parameter `path_beg`, that compares the function path in the incoming request URL to the specified function path. If it matches, for example to `/function/function_1`, then the value of the `acl` variable `url_function_1` is true, and otherwise false (line 5 in listing 6.1). Similarly, to check the storage bucket required by the function, we compare the `X-FDN-Bucket` header value against the bucket name. For example, if `X-FDN-Bucket` header value matches to `bucket_1`, then the value of the `acl` variable `bucket_1` is true, and otherwise false (line 8 in listing 6.1).

In order to forward the requests to a specific backend, it is necessary to specify the `use_backend` (lines 15-19 in listing 6.1), which takes as an argument the label of the desired backend. Given the name of the `acl`, it will forward the request to the corresponding `backend` server. The `backend` defines the group of clusters toward which the requests will be forwarded. These backends are created automatically by the **Courier Control Plane** based on the *FDN Inventory Database* (consisting of clusters, functions, and data information). It uses the data mixed with various function delivery policies to construct policies tailored for each function. `default_backend` is the backend to which the requests will be forwarded if no rule is satisfied.

There are as many `backend` sections (lines 20, 26, 29, 34, 37, and 40 in listing 6.1) as the number of policies. Each corresponds to one of the load balancers of the second layer load balancing to the subsets of clusters that can execute the associated function. `balance` specifies the load balancing algorithm that will be used. We employ WRR, and for that, we have to specify here `roundrobin` and include weights in the next lines. The weights are updated automatically based on different algorithms mentioned in §6.3.2. For each `server` in the `backend` it is necessary to specify its address, port and other secondary parameters such as the `weight` in case the policy used is WRR. For example, $function_1$ is deployed in clusters 1, 3 and 4, and *bucket*_1 is present in clusters 1 and 4. So, backend `function_1_bucket_1_policy` contains clusters 1 and 4 i.e. intersection of both lists (lines 22-25 in listing 6.1).

Listing 6.1: An example configuration of *Couier Load Balancer* based on HAProxy. The `frontend` specifies the address on which the server is listening and a set of rules for requests forwarding. The `acl` tag defines a path-based routing policy for sorting the incoming requests to the corresponding backend. Each `backend` specifies the load balancing algorithm used and the set of the clusters that will receive the requests.

```
1  frontend http_front_courier
2    bind *:80
3    stats uri /haproxy?stats
4    # functions tracking based rules
5    acl url_function_1 path_beg /function/function_1
6    acl url_function_2 path_beg /function/function_2
7    # buckets mapping
8    acl bucket_1 hdr(X-FDN-BUCKET) -i bucket_1
9    acl bucket_2 hdr(X-FDN-BUCKET) -i bucket_2
10 # based on FDN discovery inventory, courier knows (An example)
11 # 1. which functions are deployed in which clusters (functions awareness)
12 # (function1: cluster1, cluster3 cluster4, function2: cluster1))
13 # 2. which buckets are present on which clusters (data awareness)
14 # (bucket1: cluster1 and cluster4, bucket2: cluster2, cluster4))
15   use_backend function_1_bucket_1_policy if url_function_1 AND bucket_1
16   use_backend function_1_bucket_2_policy if url_function_1 AND bucket_2
17   use_backend function_1_policy if url_function_1
18   use_backend function_2_bucket_1_policy if url_function_2 AND bucket_1
19   use_backend function_2_policy if url_function_2
20   default_backend default
21
22 backend function_1_bucket_1_policy # function1 backends
23   balance roundrobin
24   server cluster1 ADDRESS:PORT weight 1
25   server cluster4 ADDRESS:PORT weight 1
26 backend function_1_bucket_2_policy
27   balance roundrobin
28   server cluster4 ADDRESS:PORT weight 1
29 backend function_1_policy
30   balance roundrobin
31   server cluster1 ADDRESS:PORT weight 1
32   server cluster3 ADDRESS:PORT weight 1
33   server cluster4 ADDRESS:PORT weight 1
34 backend function_2_bucket_1_policy # function2 backends
35   balance roundrobin
36   server cluster1 ADDRESS:PORT weight 1
37 backend function_2_policy
38   balance roundrobin
39   server cluster1 ADDRESS:PORT weight 1
40 backend default # default backend
41   balance roundrobin
42   server cluster1 ADDRESS:PORT weight 1
```

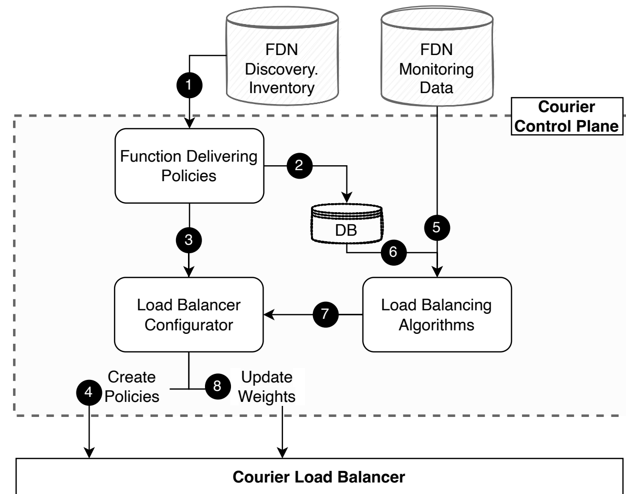


Figure 6.2.: A high-level workflow of the *Courier Control Plane*, responsible for configuring the *Courier Load Balancer* based on various function delivery policies and load balancing algorithms.

6.3 Courier Control Plane

The *Courier Control Plane* is the brain of the Courier and is responsible for configuring the *Courier Load Balancer*. A high-level workflow of the *Courier Control Plane* is shown in Figure 6.2. It is developed in Node.js programming language and stores information related to the control plane in the MongoDB database. It takes *FDN Inventory Database* (consisting of clusters, functions, and data information) (step ❶), and *FDN Monitoring Data* (consisting of monitoring metrics data from the clusters) (step ❺) as the input. It combines discovery data with various function delivery policies to construct policies tailored for each function. These policies are saved into the database of the control plane (step ❷). Simultaneously, these are passed to the *Load balancer Configurator* (§6.3.3) subcomponent (step ❸) of the control plane to create or update the configuration file of the first layer of the *Courier Load Balancer* (step ❹). *FDN Monitoring Data* (step ❺) along with the function delivery policies (step ❻) are used with various load balancing algorithms to decide the weights of each cluster within each policy. These weights are passed to the *Load balancer Configurator* subcomponent (step ❼) of the control plane to update the second layer, i.e., `backend` cluster's weights in the configuration file of the *Courier Load Balancer* (step ❽).

In the following subsections, we describe each subcomponent of *Courier Control Plane* in more detail.

6.3.1 Function Delivery Policies

Function Delivery Policies are used to *select a subset of clusters from the available clusters in FDN* . These subsets of clusters are decided based on different policies and are tailored for each function. Since the function invocations are delivered to these subsets of clusters based on the different policies, we call the policies as Function Delivery Policies. Policies are stored as JSON objects in the MongoDB database of the *Courier Control Plane*.

6.3.1.1 Function-Aware Delivery Policy

This policy considers the clusters on which a function is deployed and selects those clusters to construct policies. The *FDN Discovery Inventory* provides the function to clusters mapping. Any change in the

function's deployment, i.e., create or delete, results in a policy change. An example of a policy created for `function_1` which is deployed on `cluster_1, cluster_2 and cluster_4` is shown in listing 6.2. The object specifies the name of the function, i.e., `function_name` (line 4), which is used to create the matching criteria, and policies (lines 6-20) under the matching criteria are applicable if the matching criteria are satisfied. We can have multiple policies for the same function, but we only have one in this example. The policy contains the name (line 7), load balancing algorithm (line 8), and a set of upstreams (lines 10-20) containing the backend cluster's names, URLs, and weights. The *round_robin* algorithm is used as a load balancing algorithm. It will cause the second layer load balancer to cycle through each upstream URLs one after another with specified weights.

Listing 6.2: An example of a Function-Aware Delivery Policy created for `function_1` which is deployed on `cluster_1, cluster_2 and cluster_4`.

```
{
  "function_aware_policies":[
    {
      "function_name":"function_1",
      "policies":[
        {
          "name":"function_1_function_aware_policy_1",
          "load_balancing_algorithm":"round_robin",
          "currently_used": "true",
          "upstreams":[{
              "name":"cluster_1",
              "address":"serverless.cluster_1.fdn:31112",
              "weight":1
            },
            ...
            {
              "name":"cluster_4",
              "address":"serverless.cluster_4.fdn:31112",
              "weight":1
            }
...
```

6.3.1.2 Data-Aware Delivery Policy

FDN tracks all the data objects in MinIO storage buckets across the clusters. These buckets are the containers for data objects and support replication. The buckets' details are stored in the *FDN Discovery Inventory*. *Data-Aware Delivery Policy* uses the storage bucket required by the function, specified as the HTTP header *X-FDN-Bucket* parameter in the invocation request. It utilizes its value and gets the correct subset of target serverless compute clusters with both the bucket and the function. It uses those subsets of clusters to construct data-aware delivery policies. This policy implicitly takes function-awareness into account when selecting the subset of the clusters. FDN continuously tracks data storage buckets. Any changes (creation, deletion, and replication of bucket) are automatically taken into account by the *Courier Control Plane*, resulting in a change of the created policies. *Data-Aware Delivery Policy* has higher precedence over *Function-Aware Delivery Policy*. For instance, if `function_1` is deployed on `cluster_1, and cluster_2` and storage bucket `bucket_1` exists on `cluster_1` and storage bucket `bucket_2` exists on `cluster_1, and`

`cluster_2`, the generated data-aware policies are shown in listing 6.3. The only difference in this generated policy as against the *Function-Aware Delivery Policy* is that we use both the `function_name` (line 4) and `bucket_name` (line 5) to create the matching criteria and attach policies to it. One can see that the number of policies generated could get very large if we have a high number of functions and buckets. Furthermore, finding the right policy based on the matching criteria could get slower. One way to avoid this is to scale the number of load balancer instances, each load balancer mapping to a function. However, this scalability is beyond the scope of this dissertation.

Listing 6.3: An example of a Data-Aware Delivery Policy created for `function_1` which is deployed on `cluster_1, cluster_2`. Storage bucket `bucket_1` exists on `cluster_1` and storage bucket `bucket_2` exists on `cluster_1, and cluster_2`

```
1 {
2   "data_aware_policies":[
3     {
4       "function_name":"function_1",
5       "bucket_name": "bucket_1",
6       "policies":[
7          { "name": "function_1_bucket_1_data_aware_policy_1",
8           "load_balancing_algorithm":"round_robin",
9           "currently_used": "true",
10          "upstreams":[{
11               "name":"cluster_1",
12               "address":"serverless.cluster_1.fdn:31112",
13               "weight":1
14        ...
15    {
16      "function_name":"function_1",
17      "bucket_name": "bucket_2",
18      "policies":[{ "name": "function_1_bucket_2_data_aware_policy_1",
19          "load_balancing_algorithm":"round_robin",
20          "currently_used": "true",
21          "upstreams":[{
22               "name":"cluster_1",
23               "address":"serverless.cluster_1.fdn:31112",
24               "weight":1
25             },{
26               "name":"cluster_2",
27               "address":"serverless.cluster_2.fdn:31112",
28               "weight":1
29             }]
30 ...
```

6.3.2 Load Balancing Algorithms

Load balancing algorithms are used for load balancing across the clusters within each policy in the second layer of the *Courier Load Balancer*. Generally, load balancing algorithms can be categorized into two

categories: 1) Static load balancing and 2) Dynamic algorithms. **Static load balancing** algorithms derive their decisions based on pre-defined parameters, which do not get updated. Static algorithms offer low computational overhead with decent results [266]. However, most static algorithms do not include overload protection [104]. Popular load balancers such as the AWS Elastic Load Balancer [35] and the NGINX load balancer rely on static algorithms [22, 207]. The most prominent static algorithms are the RR and WRR algorithm. RR distributes the requests equally to all available targets, whereas WRR assigns requests to targets in a rotating manner by respecting pre-defined weights for each target. In general, static load balancing algorithms have the disadvantage that they do not directly react to changes in the runtime behavior of functions or the load on the target [104].

Dynamic algorithms take the state of the target systems into account when scheduling [104]. By respecting the system state, the algorithms try to prevent the overloading of the target systems. This comes at the cost of increased overhead and network communication [293]. *Least-connection*-based algorithms count the number of open connections to a target. They then try to keep these in balance (e.g., according to a pre-defined weight in the *Weighted Least Connection* algorithm [293]). Another dynamic approach is the prediction of the future workload, as shown by Lavanya et al. [167]. This prediction can also help to improve the overall energy efficiency, as unused machines could be turned off. Ren et al. use a prediction-based approach to improve the performance of the *Weighted Least Connection* algorithm [247]. Tong et al. present an algorithm that calculates the *residual load rate* of each server [293]. This rate indicates the remaining capacity of that specific target. The algorithm groups targets with similar *residual load rates* and distributes the requests in a WRR fashion between them.

We do support by default RR and *Least-connection* algorithm, but we also created a few more to reflect the changes in the runtime behavior of functions and the load on the clusters. We base the design of our algorithms on the WRR algorithm, as it offers better performance than *Min-Min*-based algorithms while having a very low overhead [266]. The weights within each policy are independently updated based on the designed algorithms. The following subsections provide more details on the two designed algorithms.

6.3.2.1 Latency-Aware Load Balancing Algorithm

The first approach we consider is a simple greedy approach, where we adapt the weights according to the functions' execution time in the target clusters. This is done to reflect the latency for each function invocation within the clusters and automatically take into account the computational capability of the cluster and available free resources. It initially assigns equal weights to all the target clusters and periodically updates them to reflect changes in the target clusters. The average execution time (measured in milliseconds) of the functions within a cluster is used as the main metric for weight estimation. We use this metric to prevent overloading of the target clusters, similar to the *Least Connection*-based algorithms. The pseudocode for the algorithm is shown in Algorithm 1. The algorithm represents a dynamic version of a WRR algorithm as it adapts its weights according to the functions' execution time in the target cluster. The algorithm takes the average execution times of the function across a specific time delta δ for each cluster as the input. It then determines the maximum execution time from the input average execution times (Line 3). Based on this, it calculates the weight of each cluster by dividing the maximum execution time by the function's average execution time on that particular cluster (Line 5-7). Then, the weights of each cluster are normalized by using a maximum weight provided by the user (Line 9-11). We can also replace the average execution times of the function metric with some other, like percentile 90 or maximum execution times of the function.

It is to be noted that we have specified the minimum weight of a cluster to be one (Line 1). Since we are building a distributed multiple clusters system, we do not want to exclude a cluster by specifying its weight

Algorithm 1. Latency-Aware Load Balancing Algorithm

```
Input: avg_exec_times = [], max_sum_weights, D          // D is number of clusters
Output: W = []                                          // weights for each cluster
1  min_weight = 1, weights_sum = 0
2  W = [1,1,..1]                                        // equal weights for each cluster
3  max_exec_time = Max(avg_exec_times)
4  for i ∈ D do
5  |  t_i = avg_exec_times_i           // function's average execution time on ith cluster
6  |  w_i = max_exec_time / t_i                         // Calculate weight for ith cluster
7  |  weights_sum = weights_sum + w_i
8  end
   // Normalise weight for each cluster
9  for i ∈ D do
10 |  w_i = max(floor((w_i / weights_sum) × max_sum_weights), min_weight)
11 end
12 return W
```

to zero. The weakest cluster in FDN can have the lowest weight as one, and it will always receive a small part of the load.

Another critical analysis point is the time delta δ, for which the average of the metrics is computed. Since the time resolution for collecting a metric in the FDN is one second, we decided to assign it one second.

6.3.2.2 SLO-Aware Load Balancing Algorithm

In this algorithm, we consider the SLO defined by the user for each function, along with the execution time. If the execution time on a cluster goes beyond the defined SLO, then the weight of the cluster will be defined as zero. Otherwise, it is calculated in the same fashion as in the *Latency-Aware Load Balancing Algorithm*.

The pseudocode for the algorithm is shown in Algorithm 2. The algorithm again represents a dynamic version of a WRR algorithm, as it adapts its weights according to the functions' execution time in the target cluster. The algorithm takes the average execution times of the function across a specific time delta δ for each cluster as the input. It then determines the maximum execution time from the input average execution times (Line 3). If the function's average execution time on a cluster is less than the defined SLO (`slo_exec_time`), then the algorithm calculates the weight of the cluster by dividing the maximum execution time by the function's average execution time on that particular cluster (Line 5-9). Otherwise, the cluster is assigned the weight of zero. Then, the weights of each cluster are normalized by using a maximum weight provided by the user (Line 11-13).

6.3.3 Load Balancer Configurator

This component is the connector between *Courier Load Balancer* and *Courier Control Plane*. It has two responsibilities. First, it takes the developed JSON policies (§6.3.1) and convert them into the configuration file required by the *Courier Load Balancer* (see listing 6.1). Second, it gets the weights from the load balancing algorithms for each cluster within each policy and updates them. It is developed in Node.js, with API endpoints for other components to call whenever an update in the configuration file is required.

Algorithm 2. SLO-Aware Load Balancing Algorithm

```
   Input: avg_exec_times = [], slo_exec_time, max_sum_weights, D   // D is number of clusters
   Output: W = []                                                  // weights for each cluster
 1 min_weight = 0, weights_sum = 0
 2 W = [0,0,..0]                                                   // zero weights for each cluster
 3 max_exec_time = Max(avg_exec_times)
 4 for i ∈ D do
 5     t_i = avg_exec_times_i              // function's average execution time on ith cluster
 6     if t_i ≤ slo_exec_time then
           // if execution time is less than the defined SLO
 7         w_i = max_exec_time / t_i                               // Calculate weight for ith cluster
 8         weights_sum = weights_sum + w_i
 9     end
10 end
   // Normalise weight for each cluster
11 for i ∈ D do
12     w_i = max(floor((w_i / weights_sum) × max_sum_weights), min_weight)
13 end
14 return W
```

6.4 Summary

In summary, the Courier component of the FDN is responsible for delivering and load balancing user's functions invocations across the edge-cloud continuum in FDN. It consists of two components: *Courier Load Balancer* (§6.2) and *Courier Control Plane* (§6.3). *Courier Load Balancer* is based on HAProxy and is composed of two layers. The first is the access point from the outside to the architecture, offering an HTTP endpoint. Successively, the function requests are delivered to the second layer using two function delivery policies: 1) Function-Aware, and 2) Data-Aware (§6.3.1). Each function delivery policy selects a subset of the available clusters and employs a load balancer for load balancing the incoming requests to the selected backend clusters. The load balancer can use either of the two described load balancing algorithms: 1) Latency-Aware and 2) SLO-Aware (§6.3.2). The *Courier Control Plane* is the brain of the Courier and is responsible for configuring the *Courier Load Balancer*. It takes *FDN Inventory Database*, and *FDN Monitoring Data* as the input. It combines discovery data with two function delivery policies to construct policies tailored for each function. *FDN Monitoring Data* along with the function delivery policies are used with the two load balancing algorithms (Latency-Aware and SLO-Aware) to decide the weights of each cluster within each policy. These decided weights are used to update the cluster's weights in the configuration file of the *Courier Load Balancer*.

7

SLAM: SLO-Aware Memory Optimization of Serverless Applications in FDN

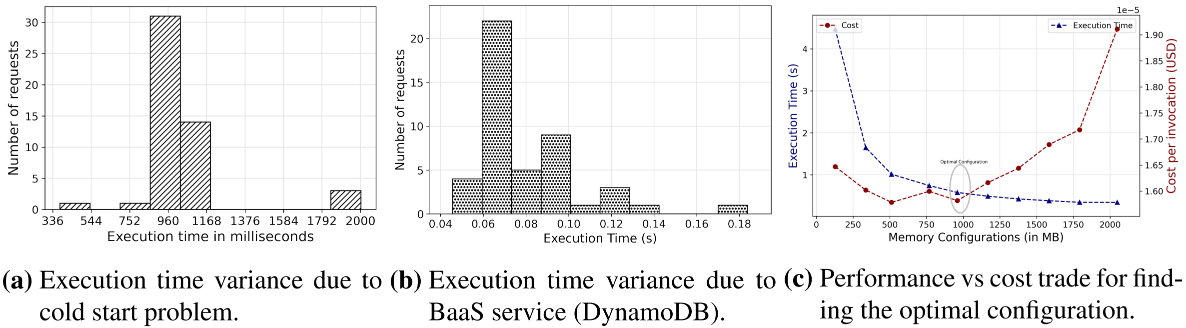
“If you work on something a little bit every day, you end up with something that is massive."

— Kenneth Goldsmith

In this chapter, we present **SLAM** (§7.2) tool, used for finding the optimal memory configuration for a serverless application when deployed on a serverless compute cluster in FDN, consisting of several FaaS functions based on the specified SLOs. Currently **SLAM** only works on the clusters based on AWS Lambda as the serverless compute platform. However, **SLAM** can be easily extended to support other commercial and open-source serverless compute platforms. We start with the motivation behind developing the **SLAM** tool in §7.1. In §7.2, we introduce **SLAM** along with its components in detail. In §7.3, we present the **SLAM** evaluation on 3 different aspects: 1) Estimation time accuracy (§7.3.2.1), 2) Configuration finding accuracy (§7.3.2.2), and 3) Configuration finding efficiency and scalability of *SLAM* (§7.3.2.3). From the experimental evaluation, the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs.

7.1 Introduction

Despite having many advantages, serverless computing suffers from some pain points that obstruct its wide adoption [47, 149, 98]. The most commonly known is optimally configuring the memory of the FaaS functions within the application based on the required the SLO. While most infrastructure management is abstracted away from the user, major commercial FaaS providers still require users to manually configure the amount of memory allocated to the FaaS functions [276]. For most developers, this is often done using their experience and knowledge, leading to suboptimal function performance and higher execution costs. The difficulties in allocating the right memory lie in the following aspects:



(a) Execution time variance due to cold start problem.

(b) Execution time variance due to BaaS service (DynamoDB).

(c) Performance vs cost trade for finding the optimal configuration.

Figure 7.1.: Various factors making it difficult to optimally configure the memory of the FaaS functions.

- **Cold start**: It is mainly connected with loading the FaaS function into the main memory of the executing server and preparing the execution environment for the target code (starting up the VM/container, loading libraries, loading function code, etc.) [202, 63]. The cold start phenomenon combined with the heterogeneity of the cloud environment makes the function execution time quite unpredictable. Figure 7.1a shows an execution time distribution for a sample compute-intensive function having a high variance when deployed with 128MB memory configuration on AWS Lambda.

- **FaaS functions integration with BaaS services**: The FaaS functions are usually closely integrated with other services, e.g., cloud databases, authentication and authorization services, and messaging services. These services are called Backend-as-a-Service (BaaS) [166]. These services also influence the execution time of the FaaS functions, thus adding the variance in the time. Figure 7.1b shows an execution time distribution for a sample function querying DynamoDB having a high variance when deployed with 128MB memory configuration on AWS Lambda.

- **Trade-off analysis between performance and cost**: Users need to define memory configuration for their FaaS functions: a low-level information that directly influences the performance and cost of the serverless application [37, 276, 302]. Thus, the user has to do a trade-off analysis between them to define the right configuration for their required SLOs [103]. Figure 7.1c shows an execution time vs the cost graph for a sample compute-intensive function when deployed with different memory configurations on AWS Lambda. It is not trivial to find the optimal configuration where the overall cost and execution time are both optimal.

- **Complex application workflows**: Usually, the serverless applications comprise dozens if not hundreds of small FaaS functions, which connect to form complex event-driven workflows. Furthermore, the SLOs are usually defined at the application level instead of the function level. Thus, based on the required application SLOs, configuring the memory of the FaaS functions within the application even becomes more challenging since a change in one can influence the others.

The aspects above highlight some factors that make it difficult for the users to optimally configure memory for serverless applications based on the required SLOs. However, there are many other factors, such as I/O and network bandwidth, and co-location with other functions affecting the performance and cost, which the users are not aware of [302]. Many researchers have addressed the issue of optimizing the memory and cost for meeting SLO requirements for a single cloud function [6, 99, 97, 321]. However, there has been a gap in solving the same problem for a serverless application consisting of many FaaS functions, which create a complicated workflow of function calls. To this end, we develop **SLAM**: SLO-Aware Memory Optimization, a python-based tool that can automatically find the optimal memory configurations for the FaaS functions within the given serverless application based on the specified SLO [253].

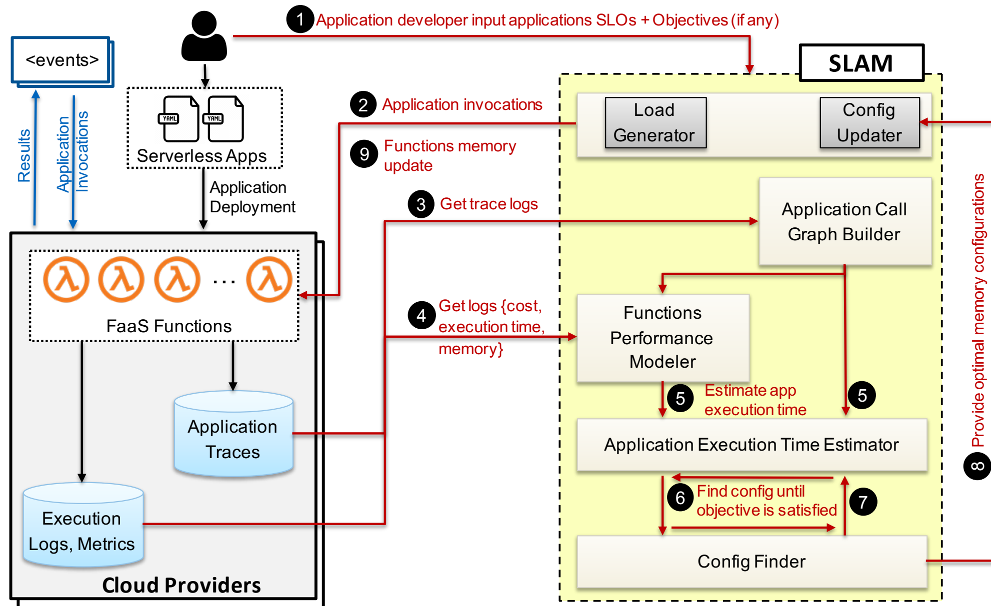


Figure 7.2.: High-level architecture of the *SLAM* and the interaction between its components.

7.2 SLAM Tool

In this section, we present *SLAM*, a python-based tool for automatically configuring the FaaS functions within a serverless application with optimal memory such that the overall execution time of invocations to the application conform to the defined SLO requirements. In this work, we consider the 95th percentile execution time of an application invocation as the SLO. *SLAM* also supports additional user-specified objectives on top of the SLO requirements: 1) Minimum Overall Cost (MOC), and 2) Minimum Overall Execution Time (MOET), by which the *SLAM* suggested configuration for the serverless application not only conforms to the defined SLO requirements, but also meets user-specified objectives. *SLAM* can dynamically adapt to changes in the given serverless application and automatically adjust memory configurations of functions. *SLAM* can be incorporated into a serverless compute platform and then leveraged by application developers for optimizing the memory configuration of their serverless applications.

Figure 7.2 provides an overview of our developed *SLAM* tool and the interaction between its components in a typical usecase. *SLAM* assumes that the serverless application which is to be configured is already deployed by the application developer on a serverless compute platform (AWS Lambda [37] in our case). Additionally, it is instrumented with a middleware tracing library (such as AWS X-Ray [33]) to trace incoming and outgoing requests to other functions or cloud components/services.

SLAM takes the SLOs requirement for the application as the input along with other user-specified objectives (if any) (step ❶). Then the *Load Generator* component of it generates a minimal amount of user workload ($K = 50$ application invocations, see Table 7.1) to the application's public endpoint (step ❷) and collects application trace logs (step ❸) and various monitoring metrics data (step ❹). The collected logs are used by the *Application Call Graph Builder* component to construct the application call graph (step ❸). This call graph and the monitoring metrics data are further used by the *Functions Performance Modeler* component for building the application's functions performance models. *Application Execution Time Estimator* component uses models along with the application call graph for estimating the overall application response time on the different memory configurations provided by *Config Finder* component. *Config Finder* component

Table 7.1.: Symbols and definitions used in the context of *SLAM* tool.

Symbol	Interpretation
N	total number of functions in an application
M	total number of memory configurations
S	total number of sequence groups formed from application call graph
U_i	total number of sub-sequence groups within some group i
K	total number of user-requests for load generation
X	possible number of memory configurations adhering to the defined SLOs.
m_i^j	memory allocated to i^{th} function in the jth configuration set
mem_config_list	a list of memory values [128, 256, 512, 1024, 2048, 4096, 8192, 10240]
$F = \{f_1, \dots, f_N\}$	functions within an application
$G = \{g_1, \dots, g_S\}$	sequence groups from application's call graph
$\bar{G} = \{\bar{g}_1^i, \dots, \bar{g}_U^i\}$	sub-sequence groups within some group i
$C = \{C_1, \dots, C_X\}$	memory configs adhering to the defined SLOs.
α	n^{th} percentile (called choice percentile) of the distribution as a representative for the execution time for the given function at a particular memory configuration.

generates the configuration based on the developed algorithms (§7.2.5.2) and examine the estimated time, memory configurations, and cost for the SLOs requirements and user-specified objective (if any) satisfaction (step ❻). If the SLOs requirements and user-specified objective are not satisfied, *Config Finder* tries different memory configurations (step ❼) and continues the process until it is satisfied (steps ❻ - ❼). Once a configuration is found, the functions' memory configurations are updated by *Config Updater* component (steps ❽ - ❾).

Next, in more detail, we describe the six major components of *SLAM* tool.

7.2.1 Load Generator

This component is responsible for generating user workload for the deployed application. It takes a total number of requests to the application as input and, based on it, generates the given amount of user workload requests synchronously to the deployed application. This user workload generation allows the creation of application traces and collection of various metrics data used by the other components of the *SLAM*.

7.2.2 Application Call Graph Builder

This component is responsible for building the application call graph involving the application functions and BaaS services such as database, storage, and queues. *SLAM* relies on external middleware tracing libraries (such as AWS X-Ray) instrumented by the application developer, allowing to trace the incoming and outgoing requests to other functions or BaaS services. The tracing library creates a "segment" for each request to the components (other functions, or BaaS services) and completes the segment as soon as the request is over. This segment describes a node in the call graph consisting of a host, request, response, start/end time, sub-segments, and errors during the process. The combination of these segments is called a

trace for a request. This component, with the help of *Load Generator* component, generates a small amount of user workload requests to the deployed application. The application traces are then parsed to generate the application call graph involving all the functions and BaaS services within the application. Afterward, the component filters out BaaS services, as it is out of the scope of this work to tune them. Moreover, it is assumed that these BaaS services provide high scalability and serve the user requests within the defined SLOs. As a result, after this step, we get the simplified call graph for the deployed serverless application and the composing functions. If the user already has the application call graph and wants to skip this step, *SLAM* allows the user to input the application's call graph manually. This also increases the testability of the *SLAM* for further development.

7.2.3 Functions Performance Modeler

After building the call graph of the application and knowing its composing functions, the next step is to estimate the execution time of each function within the serverless application at different memory configurations. This is done in two steps, explained next.

First, this component, with the help of the *Load Generator* component, generates a small amount of user workload requests ($K = 50$ application invocations, see Table 7.1) to the deployed application when all of its composing functions are deployed with a default same memory configuration (`128MB`). Based on the composing functions found by the *Application Call Graph Builder* component, it then requests the *Config Updater* component for updating the memory configurations of those functions (mem_config_list in Table 7.1). It uses the *Load Generator* to generate the same amount of user workload requests to the updated application again. The process is repeated for all the memory configurations (mem_config_list in Table 7.1). In the end, application traces and various metrics data are created to estimate the execution time for each function within the application.

Second, traces are parsed, and metrics are analyzed to create a distribution of execution time for each function and each memory configuration. An example of such a distribution for a test function, when deployed with `128MB` memory configuration on AWS Lambda, is shown in Figure 7.1a. One can observe a high variance in the execution time of the function running under the same configuration due to the uncertainties from the underneath virtualized cloud infrastructure, such as co-location of functions, cold-start, hardware failures, and resource overuse. Therefore, to overcome this inherent variance, we choose a hyperparameter called *choice percentile* (α in Table 7.1) representing the nth percentile of the distribution as a representative of the execution time for the given function at a particular memory configuration. α is configured automatically by SLAM. Calculating prediction accuracy of execution time at multiple values of α (default test values: `50, 75, 90, 99`), *SLAM* selects the one which results in a minimum mean squared error. Thus, in the end, a list of representative values for execution time for each function and memory combination is created.

7.2.4 Application Execution Time Estimator

Given the execution time of each FaaS function comprising the serverless application estimated by the *Functions Performance Modeler* at certain memory configurations, it is the responsibility of this component to combine them to estimate the overall application execution time. Function invocations in the application can either be in parallel, a sequence, or a combination of both. Therefore, from the application call graph, it first determines which functions are executed in parallel to others by using the functions' start and end timestamps available from the traces. The tool then divides all functions into sequence groups (S in Table 7.1),

where all the functions in each group are executed in parallel to other functions in the same group, and each group is executed in sequence with other groups. Since all the functions in a group are invoked in parallel, therefore to estimate the execution time of a group, we take the maximum of the execution times of all functions in the group. In the end, we sum the execution times of each group to get an estimate of the overall application execution time. Mathematically, if we have an application consisting of N functions configured with certain memory configurations and defined as $F = \{f_1, f_2, f_3, ..., f_N\}$, with them being divided into S sequence groups defined as $G = \{g_1, g_2, g_3, ..., g_S\}$, then the execution time of the whole application is given by:

$$T(G) = \sum_{x=1}^{N} F(g_x) \tag{7.1}$$

where for some group i:

$$F(g_i) = \begin{cases} max(T(\bar{g}_1^i), \ldots, T(\bar{g}_U^i)), & \text{if } g_i \neq \text{function.} \\ \text{function execution time}, & \text{if } g_i = \text{function.} \end{cases} \tag{7.2}$$

where $\bar{g}_j^i$ $(1 \leq i \leq S$ and $1 \leq j \leq U$) being the sub-sequence group within g_i and U is the total number of sub-sequence groups within g_i.

7.2.5 Config Finder

It is the responsibility of this component of *SLAM* tool, *Config Finder*, to find the right memory configurations for all functions such that the overall application execution time adheres to the defined SLOs and the specified optimization objectives (if any). We first present the two optimization objectives (§7.2.5.1) that can be used as part of *SLAM* tool in addition to the SLO requirements, and then we introduce the algorithm for finding the optimal memory configurations (§7.2.5.2).

7.2.5.1 Optimization Objectives

Suppose there are a total of X possible memory configurations set for the serverless application defined as $C = \{C_1, C_2, ..., C_X\}$ such that $C_j = \{m_1^j, m_2^j, ..., m_N^j\}$ $(1 \leq j \leq K)$ is a memory configuration set for F adhering to the defined SLOs and $m_i^j \in M$ $(1 \leq i \leq N)$ is the memory allocated to i^{th} function in the jth configuration set. Following are the two optimization objectives that can be used as part of *SLAM* tool along with the defined SLOs:

Minimum Overall Cost (MOC): Here, the idea is to find a configuration with minimum cost for each application invocation under the given SLO requirements. This is given by:

$$\min_{j \in C} Cost(j) \tag{7.3}$$

where $Cost(j)$ $(j \in C)$ is the overall application estimated cost when the application is configured with C_j configuration. Our calculation only counts for the costs associated with the function execution. It does not consider the data transfer, storage, and other costs associated with the invocation of functions. To calculate the aforementioned execution cost, we used the data provided by AWS [38]. Though they provide pricing only for a limited number of memory configurations, we interpolated the cost as there was a linear relationship between allocated memory and cost.

Minimum Overall Execution Time (MOET): The objective is to find a configuration that would result in a minimum overall execution time of the application under the given SLO requirements. This is then given by:

$$\min_{j \in C} ExecTime(j) \tag{7.4}$$

where $ExecTime(j)$ ($j \in C$) is the overall application estimated time by *Application Execution Time Estimator* when the application is configured with C_j configuration.

7.2.5.2 Optimal Memory Configuration Finding Algorithm

Now we describe the algorithm (called *SLAM-SLO*) for finding the optimal memory configuration for serverless applications such that the overall application execution time adheres to the defined SLOs. The modified version of the algorithm for optimizing on various objectives along with the SLOs is called *SLAM-SLO-Min-Cost* for MOC and *SLAM-SLO-Min-Time* for MOET. We compared our developed algorithm with the brute force (referred to as *Brute-Force*) approach, where all possible combinations of configurations for the functions within the application are generated to find the configuration that conforms to defined SLOs and the given objective. The overall complexity of this brute force approach is $O(M^N)$.

SLAM-SLO: In this approach, we leverage the max-heap data structure to find the optimal configuration which satisfies the SLO requirements. The pseudocode for the algorithm is shown in Algorithm 3. Each function's execution time at the minimum memory configuration, i.e., `128MB`, is calculated and used to construct the max-heap. We store the function's execution time at a particular configuration as the node value, and the function name and its memory configuration are further saved as the node's metadata (Line 5-8). The function with the highest execution time in a particular memory configuration will be automatically stored at the head of the max-heap tree (Line 9). We first check if this base configuration satisfies the SLO requirements. If it does, we stop the iteration and return the configuration (Line 11-13). Otherwise, in the next step, we pop the head from the max-heap (Line 14), increase its memory to decrease its execution time (Line 16) and then push the function again back to the heap with the updated memory and execution time (Line 17-20). After this update, we check if the configuration satisfies the SLO requirements. If it does, we stop the iteration and return the configuration (Line 11-13). Otherwise, we continue the process by popping the function at the head until a configuration is found. If no configuration is found, an empty dictionary is returned. The overall complexity of this approach is given by:

$$O(NM \log N) \tag{7.5}$$

This method is highly scalable and does locally optimal steps to lower the execution time of function calls.

SLAM-SLO-Min-Cost: We further modified the *SLAM-SLO* algorithm to take cost into account for finding the optimal configuration with the MOC as the objective along with the SLO requirements. Here, the algorithm uses the *SLAM-SLO* found optimal configuration as the default configuration and tries to optimize on top of it to find minimum cost configuration. In this, every time we pop the function from the head of max-heap, we check for the following inequality at the new updated memory for that function:

$$\left| \frac{\text{new_cost} - \text{old_cost}}{\text{old_cost}} \right| \leq \left| \frac{\text{old_exec_t} - \text{new_exec_t}}{\text{old_exec_t}} \right| \tag{7.6}$$

where `new_cost` and `new_exec_t` are the cost and execution time of an application invocation after updating the function's memory, and old_cost and old_exec_t correspond to the cost and execution time before the

Algorithm 3. SLAM-SLO Algorithm

```
   Input:  func_list, mem_config_list: List[ ], SLO)
   Output: result_config = Dict[str, int]
   // get minimum memory
1  min_mem_config = min(mem_config_list)
2  for func_name in func_list do
       // init minimum memory assignment for all functions
3      res_config[func_name] = min_mem_config;
4  end
   // prepare heap with function's exec time at min memory
5  for fname in func_list do
       // get exec time at min memory config for each function
6      func_exec_time = exec_time(fname, min_mem_config);
7      func_heap.append(func_exec_time, fname);
8  end
9  heapify_max(func_heap); // reorder heap
10 do
       // check for objective(s) satisfaction.
11     if estimate_exec_time(res_config) ≤ SLO then
           // other objectives can be added here.
12         return res_config;
13     end
       // get function with highest exec time.
14     top_func = heappop_max(func_heap);
15     if not all_memory_config_evaluated(top_func) then
           // update memory and time, then append to heap
16         func_new_mem = update_memory(top_func);
17         func_new_exec_time = get_exec_time(top_func, func_new_mem);
18         func_heap.append(func_new_exec_time, top_func);
19         res_config[top_func] = func_new_mem; // update
20         heapify_max(func_heap); // reorder heap
21     end
22 while func_heap is not empty;
23 return ; // return the empty config
```

update. If the inequality holds, we put the function back into the max-heap with the updated execution time. If it does not, we fix the memory for that function in the final configuration. This also allows us to reduce the search space for finding the configuration satisfying the minimum cost objective.

SLAM-SLO-Min-Time: This modified version of the *SLAM-SLO* algorithm also uses the *SLAM-SLO* found optimal configuration as the default configuration and tries to optimize on top of it to find minimum execution time configuration. It then leverages the binary search algorithm to find the configuration with minimum time. It uses the *SLAM-SLO* found optimal configuration execution time (β in seconds) as the maximum time and `0s` as the minimum time. It then updates the SLO requirement to the middle of maximum and minimum time and calls the *SLAM-SLO* algorithm to find an optimal configuration. If a configuration is found, then the maximum is set to the execution time for that configuration. Otherwise, the minimum is updated to the previously found middle. This way, it continues until a configuration is found with minimum application execution time. To avoid running the binary search indefinitely, we use a hyperparameter called precision (γ). When the lower and upper execution time bounds get closer than the precision hyperparameter, we stop the search and return the attained configuration. As a default value of the parameter, we chose $\gamma = 0.01s$, which can be easily changed.

7.3 SLAM Evaluation

We test the *SLAM* tool for serverless applications deployed on a cluster within FDN based on AWS Lambda.

7.3.1 Evaluation Settings

SLAM tool itself was run within FDN on a machine with 8 physical cores (Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz CPU) with hyperthreading enabled and 16 GB of RAM. As all the functions within our test applications use only one thread, we limit the maximum memory configuration to `2GB`. Since at that point, AWS stops increasing the portion of the allocated vCPU and increases the number of available vCPU [302], which the application will not use. For our experiments, the total number of requests for load generation is set to `50`. To test the *SLAM* tool, we have developed an interface that can automatically create synthetic applications having a different number of functions [253]. The input to the interface defines the application call tree containing functions that are either invoked in parallel or sequence. This way, we can generate complex applications with as many functions. Each function within the application is a compute-intensive function. It calculates the remainder for all numbers between 2 and N, where N is the parameter fixed for the function. The algorithm's simplicity allows us to simulate test applications with heterogeneous functions requiring different compute/memory resources by scaling N. Each function within the application has a different value for N and is assigned randomly.

7.3.1.1 Test Applications

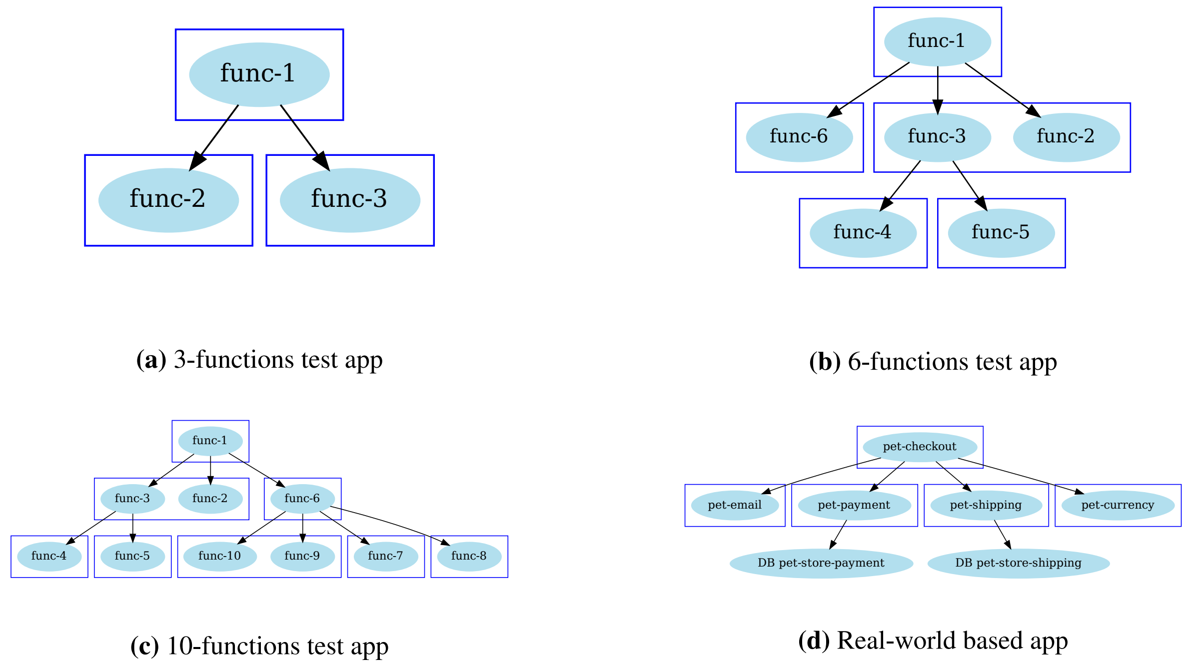
An example application callgraph with three functions where one function (func-1) is invoking the other two (func-2, func-3) in parallel is shown in Figure 7.3a. To better interpret the callgraphs, we decorated them with boxes that grouped several functions. Functions in the same box are called in parallel to each other, while the ones on the same level are called in sequence. The directed edges show the function that generates the invocation for the other functions on the lower level. The callgraph does not fully represent each function's computation, which is separate from other function calls. Those calculations are always done serially to the calls of its children's functions. We additionally created two more synthetic complex applications containing 6 and 10 functions incorporating sequence and parallel invocations to test the *SLAM* tool. Their call graphs are shown in Figure 7.3b and Figure 7.3c respectively.

Since the synthetic application workloads do not fully represent the real-world use cases for serverless applications, we created a pet store application based on an open-source spring-based application consisting of five FaaS functions and two NoSQL databases [253]. Its call graph is shown in Figure 7.3d. We used DynamoDB for the two NoSQL databases. This application is special since the functions querying databases will not influence execution time with the increase in memory. The application is a skeletal representation of what a real one would look like; it does not ship anything.

7.3.1.2 Evaluation Questions

We design our experiments to answer the questions:

- **Q1. *SLAM* estimation time accuracy**: how accurate is *SLAM* in estimating the execution time of an application for the given or found configuration at different SLOs?
- **Q2. *SLAM* configuration finding accuracy**: how accurate is *SLAM* in finding the configuration satisfying the given SLOs and objectives for an application?



(a) 3-functions test app

(b) 6-functions test app

(c) 10-functions test app

(d) Real-world based app

Figure 7.3.: Call graphs for the applications used for evaluating *SLAM*.

- **Q3. *SLAM* configuration finding efficiency and scalability**: how efficient is *SLAM* in finding the configuration satisfying the given SLOs and objectives for an application? Additionally, how does the *SLAM* tool scale with the increase in the number of functions of the application?

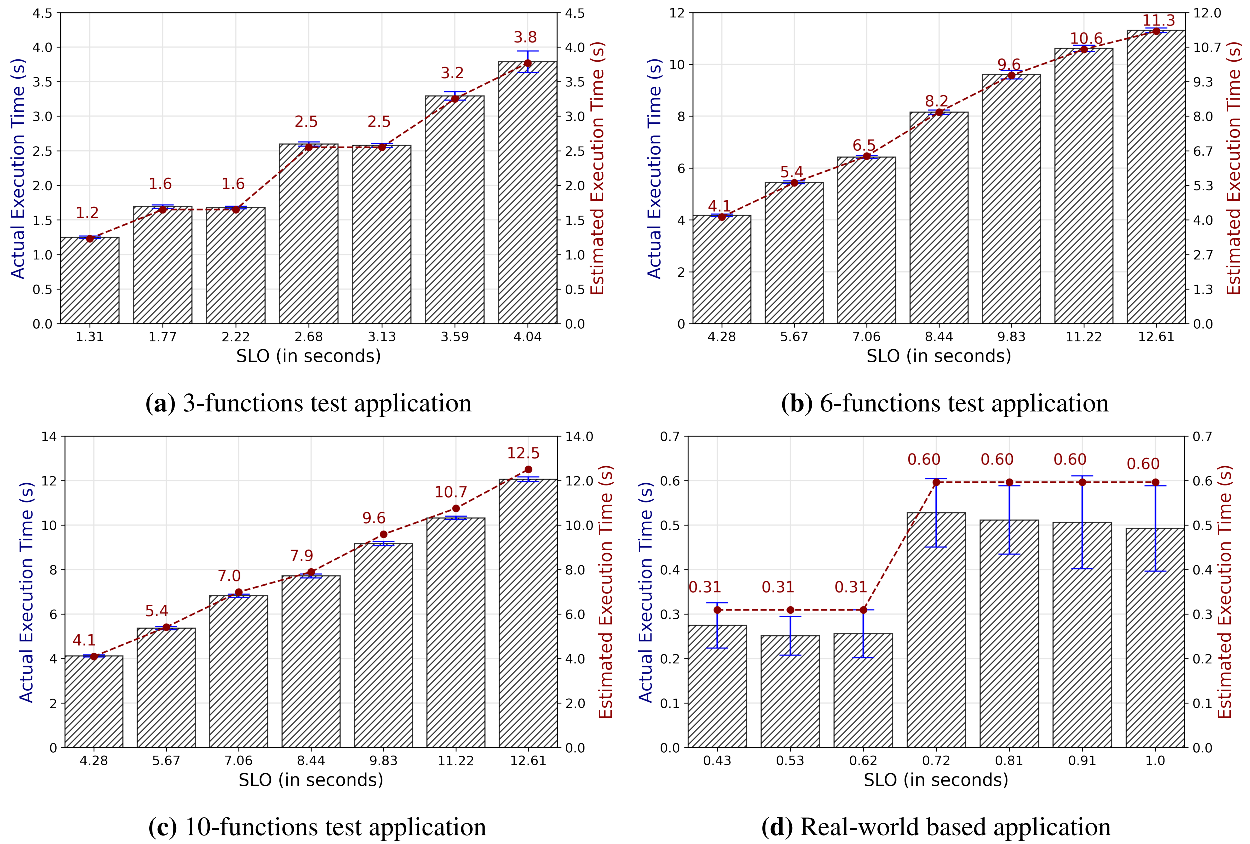
7.3.2 Results

7.3.2.1 Q1. SLAM Estimation Time Accuracy

To demonstrate the effectiveness of the *SLAM* tool in estimating the application's execution time, we test it on three synthetic and one real-world-based application. For this test, *SLAM* tool's *SLAM-SLO* algorithm is used to find the memory configurations for the given different SLOs without any additional objectives. Based on the found configuration, we configured all the functions with the memory values suggested by *SLAM-SLO* and invoked the serverless application 100 times to get the actual application's execution time distribution. Figure 7.4 shows the actual experiment execution time box plot overlaid with the estimated execution time by *SLAM-SLO* algorithm for all four test applications at different SLOs when configured with the found memory configurations. Additionally, we measured the execution time estimation accuracy percentage for the four test applications at different SLOs and is shown in Figure 7.5. For computing the accuracy at different SLOs, we calculate the mean squared percentage error between the estimated and actual execution time for the found configuration and then subtract it from 100.

Next, we discuss the results of the two classes of the test applications in more detail.

Synthetic Applications: From Figure 7.4, one can observe that in the three synthetic applications, the estimated execution time is either lower or equal to that of the specified SLOs. Additionally, from the overlaid graph of estimated execution time in Figure 7.4, we can observe that the estimated execution time



(a) 3-functions test application

(b) 6-functions test application

(c) 10-functions test application

(d) Real-world based application

Figure 7.4.: Actual execution time box plot overlaid with the estimated execution time by *SLAM* run with different SLOs.

to a great extent, is closer to the actual execution time at different SLOs. To verify it further, in Figure 7.5, the measured execution time estimation accuracy percentage for the three test applications at different SLOs is above 90%.

Real-world based Application: From the overlaid graph of estimated execution time in Figure 7.4d, one can observe that the estimated execution time is higher than the actual execution time at different SLOs. It is also evident from Figure 7.5d, where the measured execution time estimation accuracy percentage at different SLOs is lower compared to synthetic applications (ranging between 70% and 85%). However, similar to the three synthetic applications, the estimated execution time is either lower or equal to that of the specified SLOs. Thus, the configuration selected by the *SLAM* tool is good enough to fulfill the desired SLOs. One reason for the higher estimated execution time at different SLOs could be due to the involvement of components such as DynamoDB, which can lead to the variable execution time of the application. Moreover, this application's overall execution time is smaller compared to synthetic applications. Thus even the tiny inherent variance within the application can cause high relative error rates and hence the drop in the accuracy estimation. Nonetheless, as mentioned earlier, the configuration selected by the *SLAM* tool is good enough to fulfill the desired SLOs. Furthermore, from Figure 7.4d, we can observe that after a particular SLO (0.72s), the estimation and actual overall execution time for the applications almost become constant. This is because all the functions are assigned the minimum memory configuration. Therefore the overall execution time of the application is highest at that configuration and cannot go beyond it.

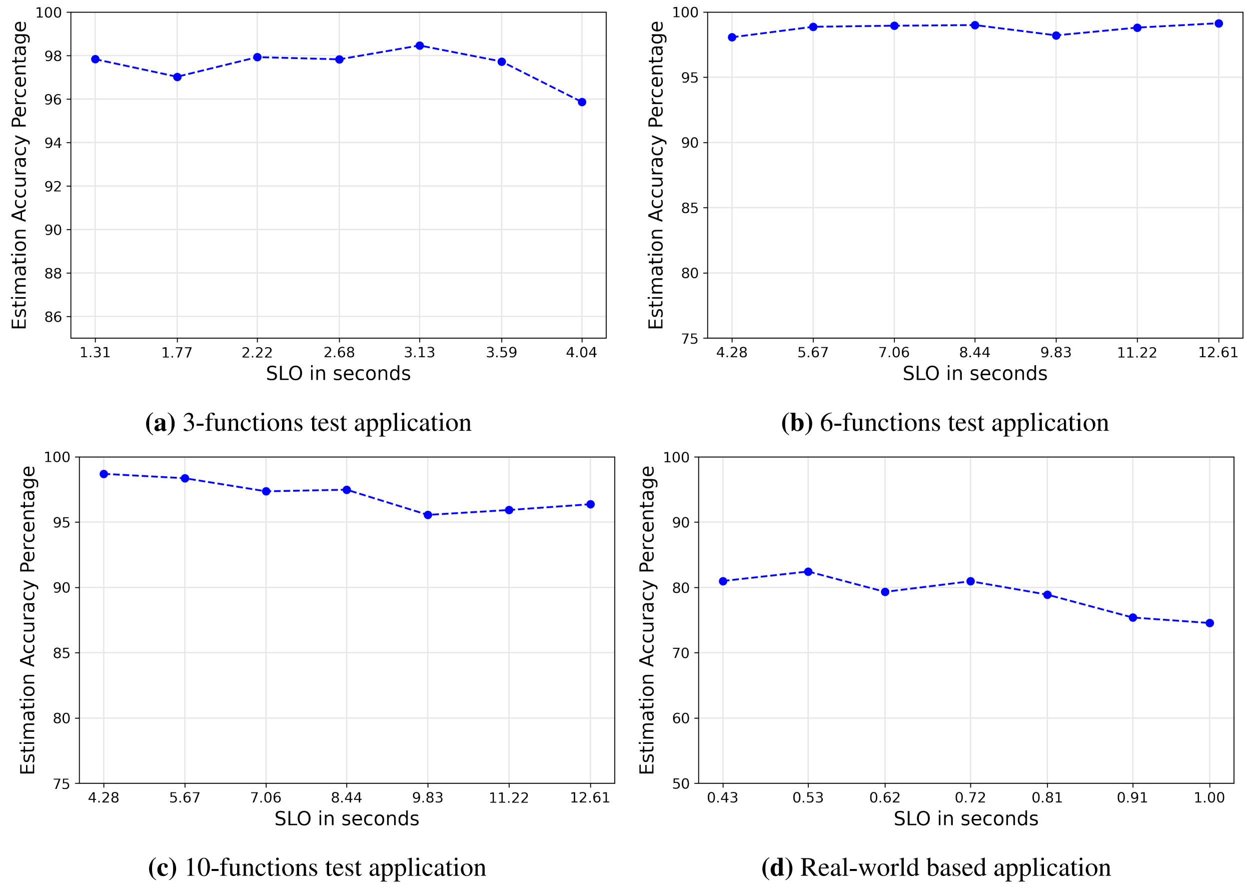


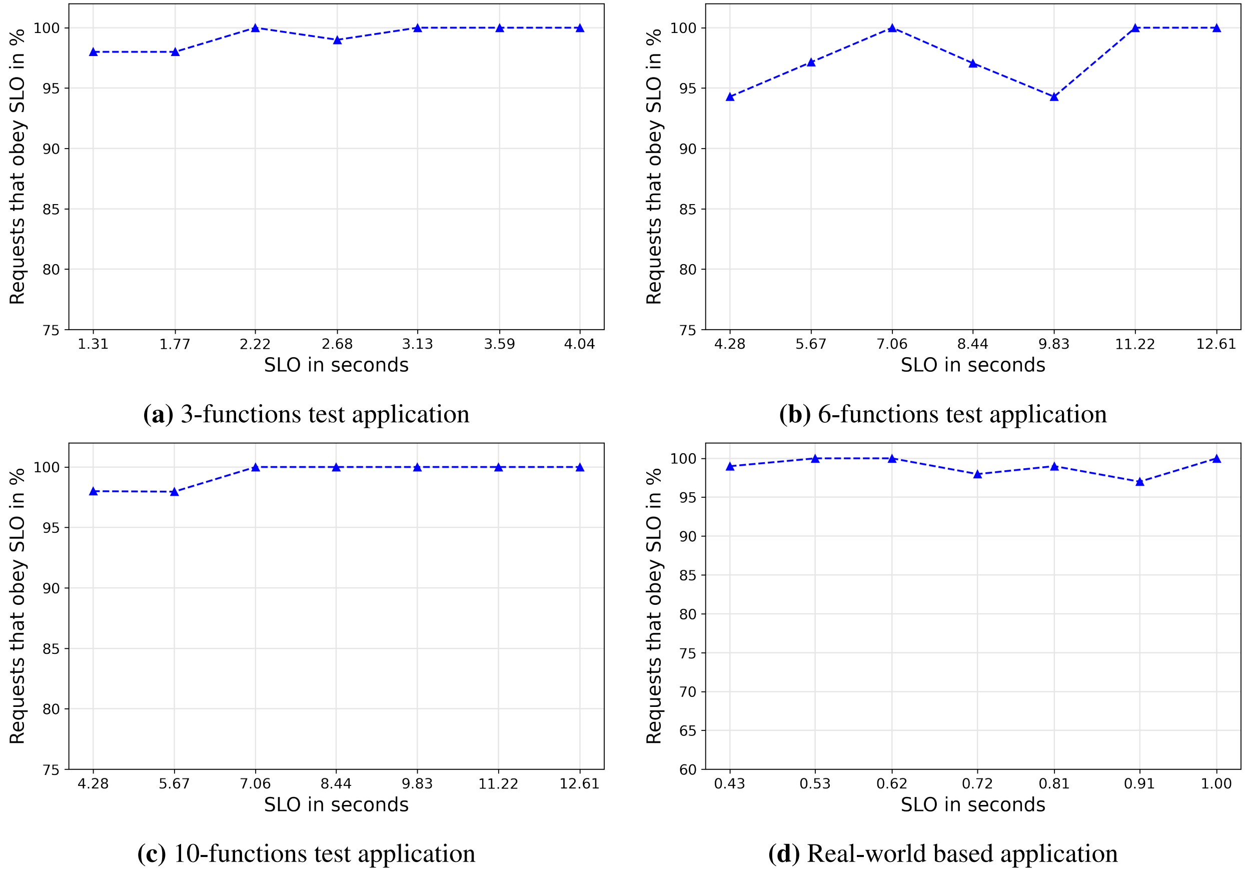
Figure 7.5.: Execution time estimation accuracy percentage for the four test applications at different SLOs.

7.3.2.2 Q2. SLAM Configuration Finding Accuracy

In this experiment, we have considered two aspects presented next for determining the accuracy of *SLAM* in finding the configuration at the given SLOs.

Precision of requests conforming SLO requirements: Here, we calculate the percentage of requests conforming to the defined SLOs when the functions are configured with the memory configurations suggested by *SLAM-SLO* algorithm. Experiment results on the four test applications are shown in Figure 7.6 for different SLOs when a total number of 100 requests were issued to the application at each SLO. We can observe that for all the synthetic applications, the percentage of requests conforming to the given SLOs is either equal to or above 95%, which means that out of issued 100 requests, at least 95 requests were served within the specified SLO execution time. Additionally, for the *Real-world based* application as well, despite having lower estimation time accuracy as compared to synthetic applications, *SLAM* is still able to generate configurations that result in above 95% precision of requests conforming to the given SLOs.

Various objectives' configuration finding effectiveness: In this aspect, we determine the effectiveness of *SLAM* tool when requested to optimize for various optimization objectives (§7.2.5.1) in addition to the SLOs. In this regard, we calculated the overall execution time and the cost needed by one application invocation when configured with memory configurations selected by *SLAM* for those optimization objectives. We compared them against static minimum-memory=128MB (*min-mem*) and maximum-memory=2GB (*max-mem*) configurations to get the worst and best execution times for the applications, and also the corresponding



(a) 3-functions test application

(b) 6-functions test application

(c) 10-functions test application

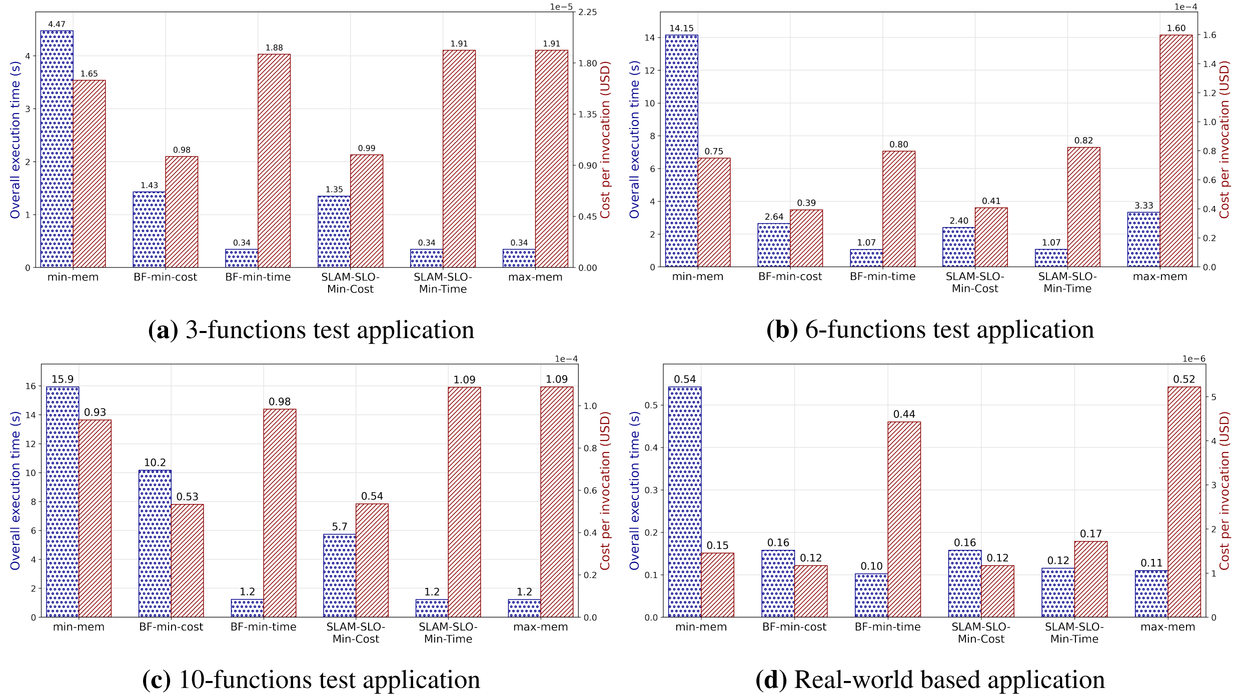
(d) Real-world based application

Figure 7.6.: Percentage of the requests conforming to the given SLOs based on the configurations suggested by *SLAM*.

costs. It is to be noted that we do not need to get the worst/best execution times at the extreme end of execution time [69]. Therefore we compare them with the global minimum cost (*BF-min-cost*) and execution time (*BF-min-time*) for each application obtained by checking every configuration and function combinations using *Brute force*. Experiment results on the four test applications are shown in Figure 7.7, and the results are averaged over 100 application invocations.

From Figure 7.7, we can see that for all the applications, *SLAM* optimization objective algorithms find the optimal/near-optimal cost and time configurations such that they are very close to the global minimum cost (*BF-min-cost*) and time (*BF-min-time*). Since the behavior of the *SLAM* on different applications is very similar, we only explain the results for the *3-functions* application on two objectives:

- **Minimum Overall Cost (MOC)**: For the *3-functions* application, *SLAM-SLO-Min-Cost* ($\$0.99 \times 10^{-5}$ as seen in Figure 7.7a) is only $\$0.01 \times 10^{-5}$ higher than *BF-min-cost* ($\$0.98 \times 10^{-5}$). When comparing *SLAM-SLO-Min-Cost* with the *min-mem* and *max-mem* configuration, *SLAM-SLO-Min-Cost* takes on average 1.6x less cost than *min-mem* and 1.9x less cost than *max-mem*. Additionally, *SLAM-SLO-Min-Cost* configuration (1.3*s*) is able to process application requests faster than the *min-mem* (4.5*s*) and *BF-min-cost* configurations (1.4*s*) but takes longer time than the *max-mem* configuration (0.3*s*).
- **Minimum Overall Execution Time (MOET)**: For the *3-functions* application, the execution time for *SLAM-SLO-Min-Time* configuration (1.07s as seen in Figure 7.7a) is equivalent to that of *BF-min-*



(a) 3-functions test application

(b) 6-functions test application

(c) 10-functions test application

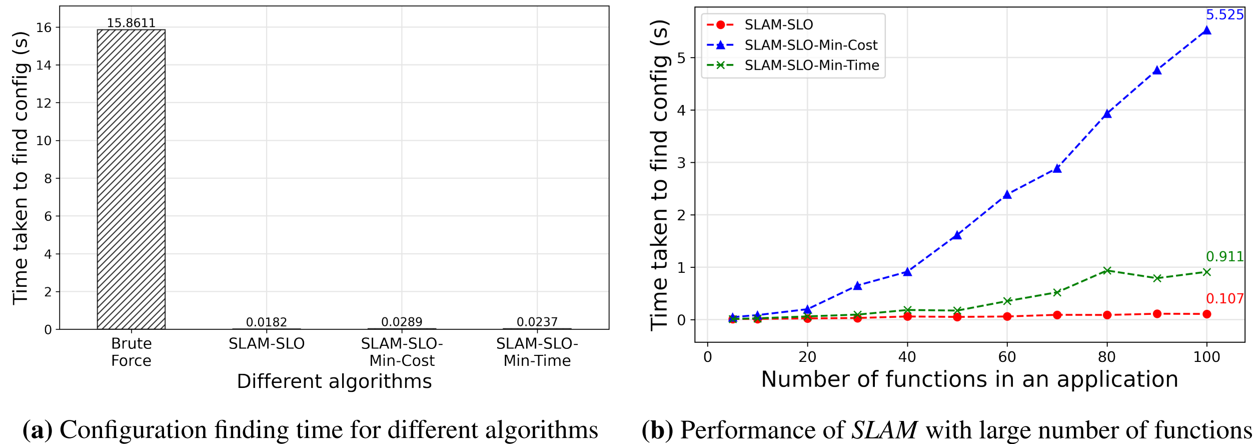
(d) Real-world based application

Figure 7.7.: Execution time and the cost when configured with configurations selected by *SLAM* for various objectives.

time configuration and the overall cost for *SLAM-SLO-Min-Time* (\$0.82 $\times 10^{-5}$) is only a bit higher than the *BF-min-time* configuration (\$0.80 $\times 10^{-5}$). This shows that *SLAM* can find the optimal/near-optimal execution time configuration such that it is very close to the global minimum execution time configuration (i.e., *BF-min-time*, which requires a long time for determination). From Figure 7.7a again we can see that the execution time taken by *max-mem* configuration (3.3s) is higher than that of *BF-min-time* configuration (1.07s), therefore it may not always be true that the largest memory results in minimum overall execution time [69]. When comparing *SLAM-SLO-Min-Time* configuration (1.07s) with the *min-mem* (14.15s) and *max-mem* (3.3s) configurations, *SLAM-SLO-Min-Time* configuration takes on average 13.5x less execution time than *min-mem* configuration and 3x less execution time than *max-mem* configuration. Additionally, *SLAM-SLO-Min-Cost* configuration (1.3s) is able to process application request faster than the *min-mem* (4.5s) and *BF-min-cost* (1.4s) configurations but takes longer time than the *max-mem* (0.3s) configuration.

7.3.2.3 Q3. SLAM Configuration Finding Efficiency and Scalability

In Figure 7.8, we show how efficient and scalable *SLAM* is in finding the optimal configurations at various objectives. Figure 7.8a shows the time required for different optimization algorithms to find the optimal configuration when run on *6-functions* application. The *Brute-force* algorithm performed worst compared to the developed optimization algorithms (almost took 871x time more than the developed algorithm). Although it is possible to parallelize the *Brute-force* search, but it is beyond the scope of this work. When comparing *SLAM-SLO* (0.0182s) with *SLAM-SLO-Min-Cost* (0.0289s) and *SLAM-SLO-Min-Time* (0.0237s), *SLAM-SLO-Min-Cost* requires the most amount of time for the application with 6 functions. This can also be validated from the Figure 7.8b where the scalability of the three algorithms is tested on applications containing a larger number of functions (from 1 to 100), and *SLAM-SLO-Min-Cost* requires the most amount of



(a) Configuration finding time for different algorithms **(b)** Performance of *SLAM* with large number of functions

Figure 7.8.: *SLAM* efficiency and scalability performance

time. All algorithms scale linearly with the number of functions in the application but with different slopes, and *SLAM-SLO* has the least slope.

SLAM-SLO-Min-Cost, which has to estimate the cost at every step of the search, has to go through a higher number of configurations as compared to *SLAM-SLO* and *SLAM-SLO-Min-Time*. Nevertheless, an application containing 100 functions *SLAM-SLO-Min-Cost* took 5.5s, which is not a lot, considering the benefits of the algorithm in terms of cost-saving.

7.4 Summary

Serverless computing has abstracted most cloud server management and infrastructure scaling decisions away from the users, but configuring the memory of FaaS functions is still left up to the users. To solve this problem, in this chapter, we introduced **SLAM** to find optimal memory configurations for a serverless application when deployed on a serverless compute cluster based on AWS Lambda in FDN, given predefined SLO requirements. **SLAM** uses a max-heap-based optimization algorithm along with its variants for various optimization objectives (minimum cost and minimum overall time) in finding the optimal memory configuration for the given serverless application based on the specified SLO. It supports complex serverless application call-graph workflows and can adapt to changes in a serverless application. In this chapter, we demonstrate the functionality of *SLAM* with AWS Lambda (§7.3) on four serverless applications consisting of a various number of functions and found that the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs.

8

Anomaly Detection in the FDN

“Life is like riding a bicycle. To keep your balance, you must keep moving."

— Albert Einstein

Serverless compute clusters within FDN are created using VMs hosted on the bare-metal server using a hypervisor. An anomaly in the application’s functions deployed in those VMs can affect the availability and reliability of the application. Furthermore, a fault or an anomaly in the hypervisor hosting the VMs can propagate to the VMs hosted on it and ultimately affect the availability and reliability of the applications running on those VMs. Therefore, identifying and eventually resolving it quickly is highly important.

Therefore, in this chapter we describe two anomaly detection algorithms for FDN: 1) Online memory leak detection in VMs using *Precog* in §8.1, and 2) Anomalous VMMs detection using *IAD: Indirect Anomaly Detection* in §8.2.

8.1 Online Memory Leak Detection

Cloud computing is widely used in industries to provide affordable and on-demand access to computing and storage resources. Physical server resources located at different data centers are split among the vVMs hosted on it and distributed to the users [138]. Users can deploy their applications on these VMs with only the required resources. This allows for the efficient usage of the physical hardware and reduces the overall cost. However, with all the advantages of cloud computing, there is a drawback of efficiently detecting a fault or an error in an application or in a VM due to the layered virtualization stack [18, 115]. A minor fault in the system can impact the application’s performance.

When deployed on a VM, an application usually requires different system resources, such as memory, CPU, and network, to complete a task. If an application mainly uses the memory for processing the tasks, then this application is called a memory-intensive application [237]. The application’s responsibility is to release the system resources when they are no longer needed. When such an application fails to release the memory

Table 8.1.: Symbols and their definitions used in the context of memory leak detection.

Symbol	Interpretation
t	a timestamp
x_t	the percentage utilization of a resource (for example memory or disk usage) of a virtual machine at time t
N	Number of data points
$x = \{x_1, x_2, ..., x_N\}$	a VM's memory utilization observations from the Cloud
T	time series window length
$x_{t-T:t}$	a sequence of observations $\{x_{t-T}, x_{t-T+1}, ..., x_t\}$ from time $t-T$ to t
U	percentage memory utilization threshold equal to 100.
C	critical time

resources, a **memory leak** occurs in the application [313]. Memory leak issues in the application can cause continuous blocking of the VM's resources, resulting in slower response times or application failure. In the software industry, memory leaks are treated with utmost seriousness and priority, as the impact of a memory leak could be catastrophic to the whole system. In the development environment, these issues are relatively easily detectable with the help of static source code analysis tools or by analyzing the heap dumps. However, memory leak detection is a challenge in the production environment running on the cloud. It only gets detected when there is an abnormality in the run time, abnormal usage of the system resources, a crash of the application, or a restart of the VM. Then the resolution of such an issue is made at the cost of compromising the availability of the application. Therefore it is necessary to monitor every application for memory leak and have an automatic detection mechanism before it occurs. However, it is a challenge to detect memory leak of an application running on a VM in the cloud without the knowledge of the programming language of the application, the knowledge of source code nor the low-level details such as allocation times of objects, object staleness, or the object references [275]. Therefore, this challenge is addressed in this work *by solely using the VM's memory utilization as the primary metric and devising a novel algorithm called **Precog** to detect memory leak.*

8.1.1 Methodology for Memory Leak Detection

This section presents the problem statement of memory leak detection and describes our proposed algorithm's workflow for solving it.

8.1.1.1 Problem Statement

Table 8.1 shows the symbols used in the context of memory leak detection.

We are given $x = \{x_1, x_2, ..., x_N\}$, a $N \times 1$ dataset representing the memory utilization observations of the VM and an observation $x_t \in R$ is the percentage memory utilization of a virtual machine at time t. This work aims to determine whether there is a memory leak on a VM such that an observation x_t at time t reaches the threshold U memory utilization following a trend in the defined critical time C. Formally:

- **Given**: a univariate dataset of N time ticks, $x = \{x_1, x_2, ..., x_N\}$, representing the memory utilization observations of the VM.

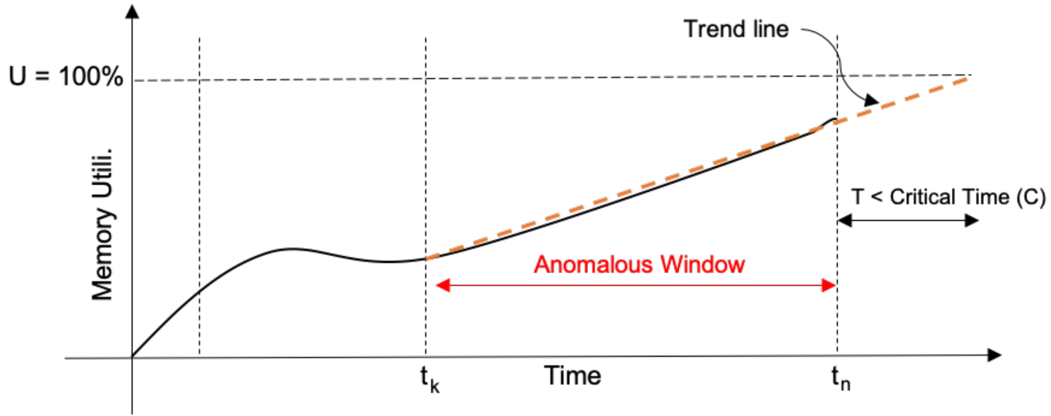


Figure 8.1.: Example memory utilization of a memory leaking VM with the marked anomalous window.

- **Output**: an anomalous window for a VM consisting of a sequence of observations $x_{t-T:t}$, such that these observations after following a certain trend will reach the threshold U memory utilization, at time $t+M$, where $M \leq C$.

Definition 1

(Critical Time) It is the maximum time considered relevant for reporting a memory leak in which, if the trend line of memory utilization of VM is projected, it will reach the threshold U.

8.1.1.2 Illustrative Example

Fig. 8.1 shows the example memory utilization of a memory leaking VM with the marked anomalous window between t_k and t_n. It shows that the memory utilization of the VM will reach the defined threshold ($U = 100\%$) within the defined critical time C by following a linearly increasing trend (shown by the trend line). Therefore, this VM is regarded as a memory-leaking VM.

Our developed approach can be applied to multiple VMs as well. We also experimented to understand the memory usage patterns of memory leak applications. We found that if an application has a memory leak, the memory usage of the VM on which it is running increases steadily. It continues to do so until all the system's available memory is exhausted. This usually causes the application attempting to allocate the memory to terminate itself. Thus, memory leak behavior usually exhibits a linearly increasing or "sawtooth" memory utilization pattern.

8.1.2 Memory Leak Detection Algorithm: Precog

The *Precog* algorithm consists of two phases: offline training and online detection. Fig. 8.2 shows the overall workflow of the *Precog* algorithm.

Offline training: The procedure starts by collecting the memory utilization data of a VM and passing it to *Data Pre-processing* module. *Data Pre-processing* module first transforms the dataset by resampling the number of observations to one for the defined resampling time resolution. Then the time series data is median smoothed over the specified smoothing window. In *Trend Lines Fitting* module, firstly, on the

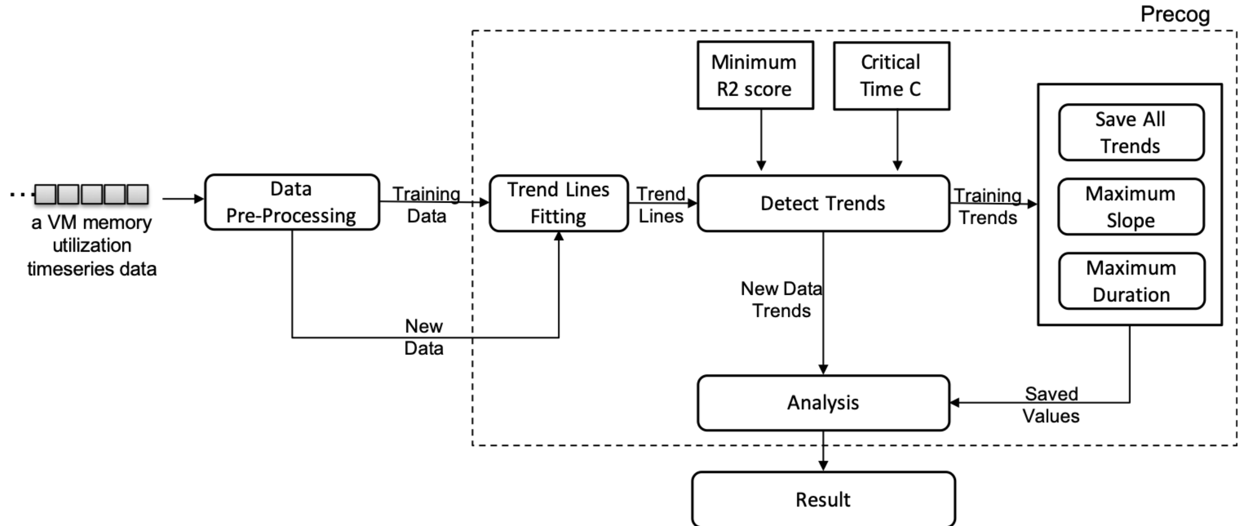


Figure 8.2.: Overall workflow of the *Precog* algorithm.

whole dataset, the change points $P = \{P_1, P_2, ..., P_k\}$, where $k \leq n-1$, are detected. By default, two change points, one at the beginning and the other at the end of time series data, are added. If the change points are not computed, the algorithm will have to go through each data point, which will be compute-intensive. Therefore these points allow the algorithm to directly jump from one change point to another and select all the points between the two change points. *Trend Lines Fitting* module selects a sequence of observations $x_{t-L:t}$ between the two change points: one fixed P_1 and other variable P_r where $r \leq k$ and a line is fitted on them using the linear regression. The R-squared score, size of the window called as *duration*, time to reach threshold called *exit time*, and slope of the line are calculated. This procedure is repeated by keeping the fixed change point the same and varying the other for all other change points. Out of all the fitted lines, the best-fitted line based on the largest duration and highest slope is selected for the fixed change point. If this best-fitted line's time to reach the threshold falls below the critical time, its slope and duration are saved as historical trends.

This above procedure is again repeated by changing the fixed change point to all the other change points. At the end of this whole procedure, we get a best-fitted trend for each change point, if it exists. Amongst the captured trends, the maximum duration and the maximum slope of the trends are also calculated and saved. This training procedure can be conducted routinely, e.g., daily or weekly.

Definition 2

(Change Points) A set of time ticks that deviate highly from the normal pattern of the data. This is calculated by first taking the first-order difference of the input timeseries. Then, take their absolute values and calculate their Z-scores. The indexes of observations whose Z-scores are greater than the defined threshold (3 times the standard deviation) represent the change points.

Online detection: In the Online Detection phase, for a new set of observations $\{x_k, x_k+1, x_k+2, ..., x_k+t-1 x_k+t\}$ from time k to t where $t-k \geq P_{min}$ belonging to a VM after pre-processing is fed into the *Trend Lines Fitting* module. In *Trend Lines Fitting* module, the change points are detected. A sequence of observations $x_{t-L:t}$ between the last two change points starting from the end of the time series is selected, and a line is fitted on them using linear regression. The R-squared score, slope, duration, and exit time to reach the threshold of the fitted line are calculated. If its slope and duration are greater than the saved maximum

Table 8.2.: Synthetically generated timeseries for each memory leak pattern and their F1-Scores.

Memory Leak Pattern	+ve cases	-ve cases	F1-Score	Recall	Precision
Linearly Increasing	30	30	0.933	0.933	0.933
Linearly Increasing(with Noise)	30	30	0.895	1.0	0.810
Sawtooth	30	30	0.830	0.73	0.956
Overall	90	90	0.9	0.9	0.91

counterparts, then that window is marked anomalous. Otherwise, the values are compared against all the found training trends, and if the slope and duration of the fitted line are greater than any saved trends, then the window will be marked as anomalous. This procedure is further repeated by analyzing the observations between the last change point P_k and the next change point until all the change points are used. This is done for the cases where the new data has a trend similar to the historical data but now has a higher slope and longer duration.

8.1.3 Precog Evaluation

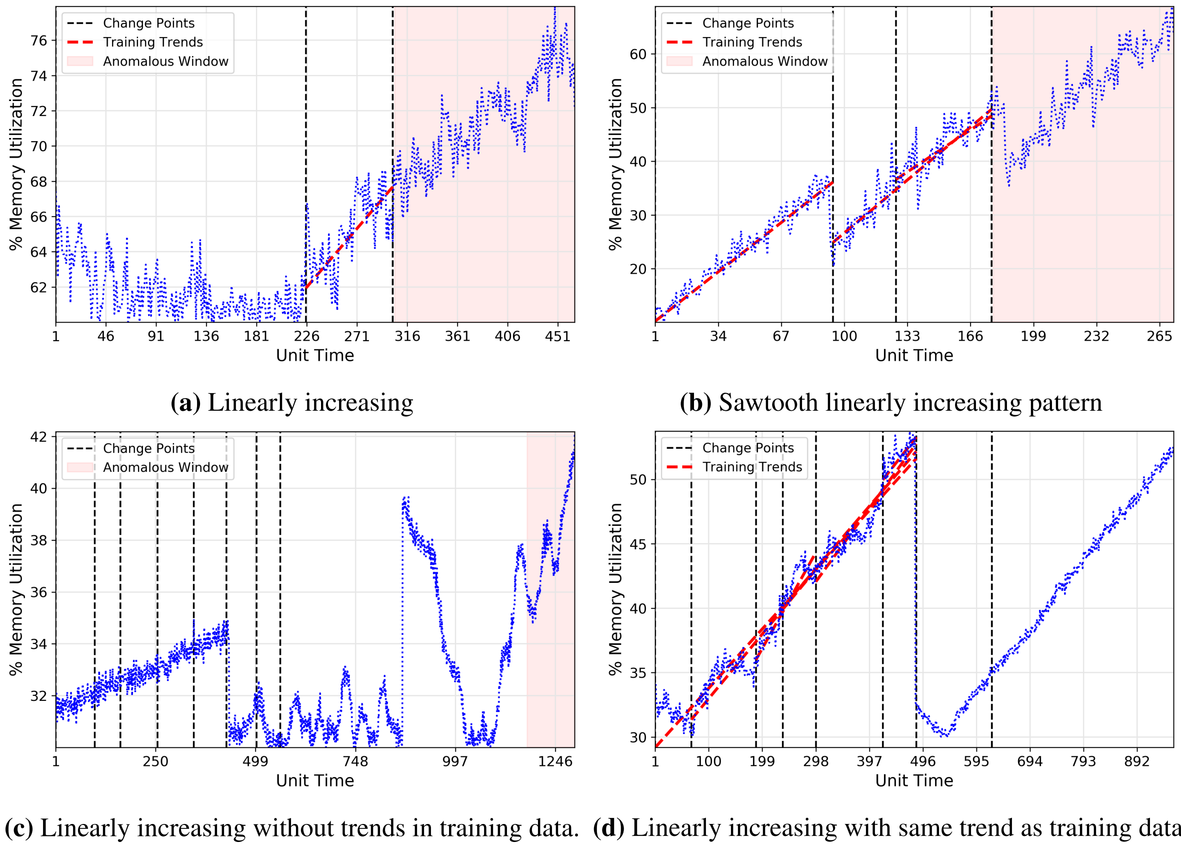
We have used F1-Score (denoted as F1) to evaluate the performance of the algorithms. Evaluation tests have been executed on a machine with four physical cores (3.6 GHz Intel Core i7-4790 CPU) with hyperthreading enabled and 16GB of RAM. These conditions are similar to a typical cloud VM. It is to be noted that the algorithm detects the cases where there is an ongoing memory leak and assumes that previously there was no memory leak. For our experiments, hyper-parameters are set as follows. The maximum threshold U is set to 100, and the defined critical time C is set to 7 days. The smoothing window size is 1 hour, and the re-sampling time resolution is set to 5 minutes. Lastly, the minimum R-squared score ($R2_{min}$) for a line to be recognized as a good fit is set to 0.75. 65% of data is used for training and the rest for testing. We design experiments to answer the following questions:

- **Q1. Memory Leak Detection Accuracy**: How accurate is *Precog* in the detection of memory leaks?
- **Q2. Scalability**: How does the algorithm scale with the increase in the data points?
- **Q3. Parameter Sensitivity**: How sensitive is the algorithm when the parameters values are changed?

8.1.3.1 Q1. Memory Leak Detection Accuracy

To demonstrate the effectiveness of the developed algorithm, we initially synthetically generated the time-series. Table 8.2 shows the F1-score corresponding to each memory leak pattern and the overall F1-score.

In addition, to demonstrate the effectiveness of the developed algorithm on the real cloud workloads, we evaluated *Precog* on the real cloud dataset provided by Huawei Munich. It consists of manually labeled memory leak data from 60 VMs spanned over five days, and each time series consists of an observation every minute. Of these 60 VMs, 20 VMs had a memory leak. Many VMs have memory leaks because applications with memory leaks were deliberately run on the infrastructure. The algorithm achieved the F1-Score of 0.857, recall equals to 0.75 and precision as 1.0. Average prediction time per test data containing approximately 500 points is 0.32 seconds. Furthermore, we present the detailed results of the algorithm on the selected four cases shown in Figure 8.3: simple linearly increasing memory utilization, sawtooth linearly increasing pattern, linearly increasing pattern with no trends detected in training data, and linearly increasing with the similar trend as training data. The figure also shows the change points, training trends, and the detected anomalous memory leak window for each case.



(a) Linearly increasing

(b) Sawtooth linearly increasing pattern

(c) Linearly increasing without trends in training data.

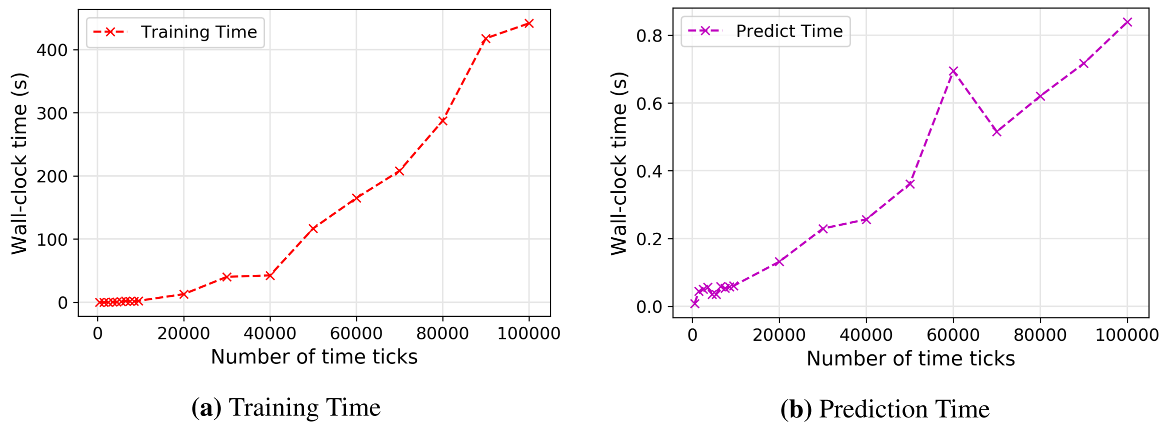
(d) Linearly increasing with same trend as training data.

Figure 8.3.: Algorithm result on three difficult cases having memory leak (a-c) and one not (d).

For the first case shown in Fig. 8.3a, memory utilization is being used normally until it suddenly increases linearly. The algorithm detected one training trend and reported the complete test set as anomalous. The test set has a similar slope as the training trend but with a longer duration and higher memory usage; hence it is reported as anomalous. In the second case (Fig. 8.3b), the trend represents a common memory leak sawtooth pattern, where the memory utilization increases up to a certain point. It then decreases (but not wholly zero), and then again, it starts to increase similarly. The algorithm detected three training trends and reported anomalous most of the test set. The test set follows a similar trend as captured during the training but with higher memory utilization; hence it is reported. In the third case (Fig. 8.3c), no appropriate training trend was detected in the complete training data. However, the algorithm detected an increasing memory utilization trend in the test dataset. In Fig. 8.3d, the VM does not have a memory leak. However, memory utilization is steadily increasing, which seems to be a memory leak pattern if observed without historical data. However, the same trend is already observed in the historical data and therefore is a normal memory utilization pattern. *Precog* using the historical data for detecting the training trends and then comparing them with the test data correctly reports that trend as normal and hence does not flag the window as anomalous. It is also to be noted that if the new data's maximum goes beyond the maximum in the training data with a similar trend, it will be regarded as a memory leak.

8.1.3.2 Q2. Scalability

Next, we verify that our prediction method scale linearly. We repeatedly duplicate our dataset in time ticks and add Gaussian noise. Figure 8.4b shows that *Precog*'s predict method scale linearly in time ticks. *Precog*



(a) Training Time

(b) Prediction Time

Figure 8.4.: *Precog*'s prediction method scale linearly.

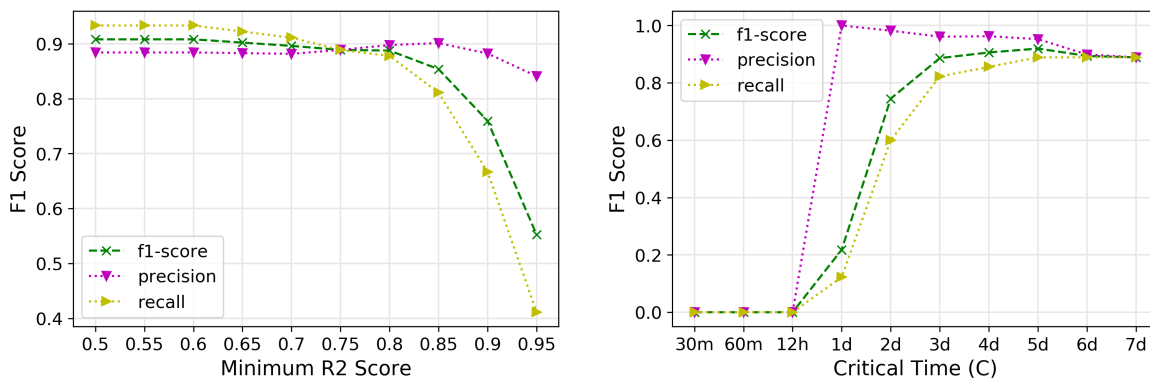


Figure 8.5.: Insensitive to parameters: *Precog* performs consistently across parameter values.

does provide the prediction results in under one second for the data with `100,000` time ticks. However, the training method shown in Figure 8.4a is quadratic, but training needs to be conducted once a week or a month and can be done offline as well.

8.1.3.3 Q3. Parameter Sensitivity

Precog requires tuning certain hyper-parameters like R2 score and critical time, which are currently set manually based on the expert's knowledge. Figure 8.5 compares performance for different parameter values, on synthetically generated dataset. Our algorithm performs consistently well across values. Setting a minimum R2 score above `0.8` corresponds to the line's stricter fitting, which is why the accuracy drops. On the other hand, our data mostly contains trend lines that would reach the threshold within three to four days. Therefore, setting the minimum critical time to less (less than three days) would mean the trend line never reaches the threshold within the time frame, decreasing the accuracy. These experiments show that these parameters play a role in the overall accuracy of the algorithm, but at most of the values, the algorithm is insensitive to them. Furthermore, determining these automatically based on historical data is out of the scope of this work.

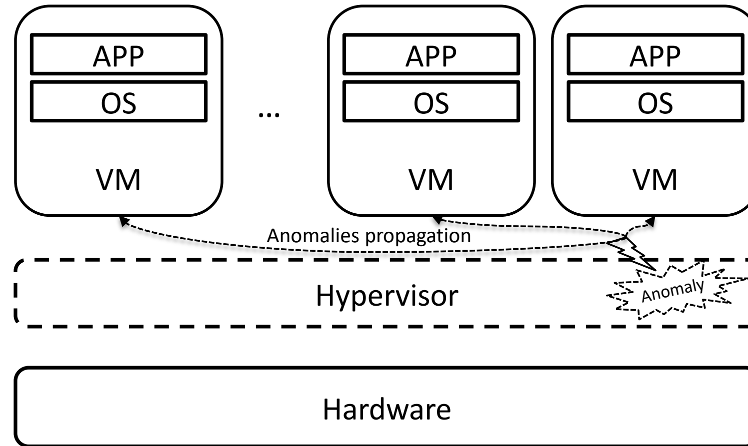


Figure 8.6.: An example showcasing the propagation of anomalies in a Type-1 hypervisor or VMM to the VMs hosted on it. These anomalies may lead to VMs failures.

8.2 Anomalous VMMs Detection

Cloud computing enables industries to develop and deploy highly available and scalable applications to provide affordable and on-demand access to compute and storage resources. Server virtualization in the form of VMs is an essential part of cloud computing technology to provide IaaS with the use of a hypervisor or VMM [230]. Users can deploy their applications on these VMs with only the required resources. This allows the efficient usage of the physical hardware and reduces the overall cost. The virtualization layer, especially the hypervisors, is prone to temporary hardware errors caused by manufacturing defects, a sudden increase in CPU utilization caused by some task, disconnection of externally mounted storage devices, etc. The VMs running on these VMMs are susceptible to errors from the underneath stack. As a result, it can impact the performance of the applications running on these VMs [168, 173]. Figure 8.6 shows an example propagation of anomalies in a virtualization stack using a type-1 hypervisor to the VM hosted on it. These anomalies may lead to the failure of all VMs and, ultimately, the applications hosted on them.

In the development environment, these anomalous VMMs are relatively easily detectable by analyzing the logs from the hypervisor dumps. However, in the production environment running on the cloud, anomalous VMMs detection is a challenge since a cloud user does not have access to the VMMs logs. Additionally, many anomalous VMM detection techniques have been proposed [315, 245, 210]. However, these either require the monitoring data of the hypervisor or injecting custom probes into the hypervisor. Therefore, the usage of such solutions becomes infeasible. Furthermore, due to the low downtime requirements for the applications running on the cloud, detecting such anomalous VMMs and their resolutions is to be done as quickly as possible.

Therefore, this challenge is addressed in this work for detecting anomalous VMMs *by solely using the VM's resources utilization data hosted on those VMMs*. We create a novel algorithm called **IAD**: **I**ndirect **A**nomalous VMMs **D**etection. We call the algorithm indirect since the detection is done without any internal knowledge or data from the VMM; it is solely based on the VMs data hosted on it.

8.2.1 Problem Definition

This section presents the overall problem definition of indirectly detecting anomalous VMMs in a cloud-based environment. Table 8.3 shows the symbols used in the context of indirectly detecting anomalous

Table 8.3.: Symbols and their definitions used in the context of indirectly detecting anomalous VMMs in a cloud-based environment.

Symbol	Interpretation
n	Number of time ticks in data
d	Number of VMs hosted on a VMM
X_t	The percentage utilization of a resource (for example, CPU or disk usage) by a VM at a time t
X_t^j	The percentage utilization of a resource at a time t for j^{th} VM
$\{c_t^1, c_t^2, ..., c_t^m\}$	a set of m $\leq$d VMs with change point at time tick t
w	Window size
minPercentVMsFault	Minimum % of total number of VMs on a VMM which must have a change point for classifying the VMM anomalous.

VMMs.

We are given $X = n \times d$ dataset, with n representing the number of time ticks and d the number of VMs hosted on a VMM. X_t^j denotes the percentage utilization of a resource (for example, CPU or disk usage) at a time t for j^{th} VM. Our goal is to detect whether the VMM on which the d VMs are hosted is anomalous or not. Formally:

- **Given** *a multivariate dataset of n time ticks, with d VMs (X_t^j for $j = \{1, \cdots, d\}$ and $t = \{1, \cdots, n\}$) representing the CPU utilization observations of VMs hosted on a VMM.*
- **Output** *a subset of time ticks or a time tick where the behavior of the VMM is anomalous.*

One of the significant challenges in this problem is the online detection, in which we receive the data incrementally, one time tick for each VM at a time, i.e., $X_1^j, X_2^j, \cdots$, for the jth VM. As we receive the data, the algorithm should output the time ticks where the behavior of the VMM is observed as anomalous. However, without looking at the future few time ticks after time t, it would be impractical to determine whether at time point t, the VMM is anomalous or not. Since the time ticks $t+1, t+2, \cdots$ are essential in deciding whether an apparent detection at time t was an actual or simply noise. Hence, we introduce a window parameter w; upon receiving a time tick $t+w$, the algorithm outputs whether at time t the VMM showcased anomalous behavior or not. Additionally, the change points for VMs hosted on VMM could be spread over a specific duration due to the effect of the actual fault being propagating to the VMs and the granularity of the collected monitoring data. Therefore, an appropriate window size can provide a way to get those change points.

8.2.1.1 Illustrative Example

Here we illustrate the problem with two examples in Fig. 8.7 showcasing the CPU utilization of two VMs hosted on a VMM. In the left sub-figure, an application runs only on VM 2, while in the right, an application runs on both VMs. During the application run time, an anomaly, i.e., high CPU load, was generated on the hypervisor for some time (shown by dotted red lines). During this time, we can observe a significant drop in the CPU utilization of the application (affecting the application's performance) of the two VMs. The load on a VMM effects all or most of the VMs hosted on it. It can significantly affect the performance of the applications running on the two VMs; therefore, we call such a VMM anomalous when the load was generated on it.

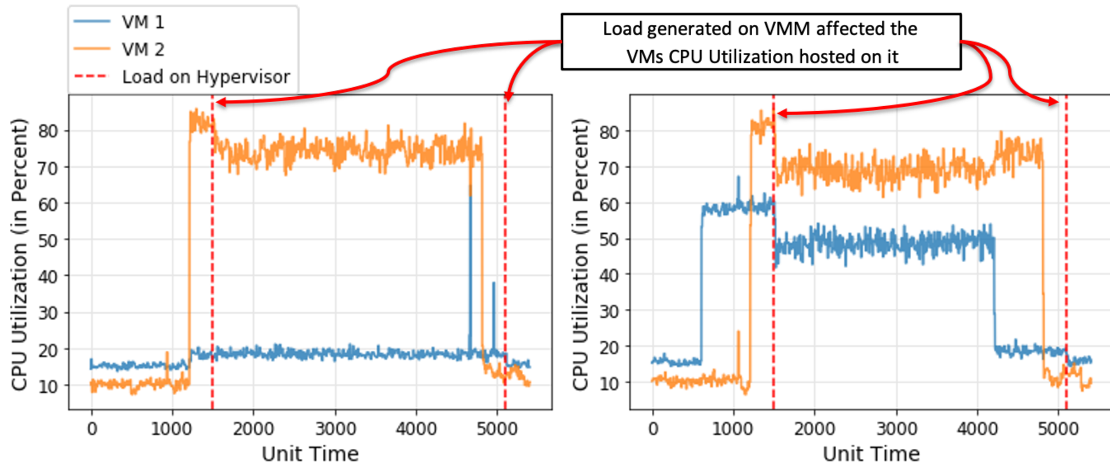


Figure 8.7.: Examples showing CPU utilization of two VMs hosted on a VMM. The left sub-figure shows an application running only on VM 2, while the right sub-figure shows the application running on both VMs. We can see a significant decrement in the CPU utilization of the two VMs when an anomaly (high-CPU load) is generated on the VMM (shown by dotted red lines).

8.2.2 Indirect Anomaly Detection (IAD) Algorithm

This section presents our proposed Indirect Anomaly Detection (IAD) algorithm and the system for evaluating it. The overall system workflow diagram is shown in Figure 8.9 and mainly consists of two parts: the main *IAD* algorithm and the *Test Module* for evaluating the algorithm.

8.2.2.1 IAD Algorithm

Our principal intuition behind the algorithm is that if a time tick t represents a change point for some resource utilization (such as CPU utilization) in most VMs hosted on a VMM; then the VMM is also anomalous at that time tick. This is based on the fact that a fault in VMM will affect most of the VMs hosted on it, and therefore those VMs would observe a change point at a similar point in time (in the chosen window w (Table 8.3)) in their resource's utilization. IAD algorithm consists of two main parts, described below:

Change Points Detector: We first explain how the change point, i.e., time tick where the time series changes significantly, is calculated. Recall from §8.2.1 that, we have introduced a window parameter w, upon receiving the time tick $t+w$, the *Change Points Detector* outputs whether the time tick t is a change point or not. Given a dataset X^j of size w for j^{th} VM, this component is responsible for finding the change points in that VM. This can be calculated in two ways: Mean-based detector and Z-score-based detector.

- **Mean-based Detector**: In this detector, a *windowed_mean*, i.e., the mean of all the values in the window, and the *global_mean*, i.e., the mean of all the values until the current time tick is calculated. Since the IAD algorithm is designed for running it online, not all the values can be stored. Thus *global_mean* is calculated using Knuth's algorithm [160, 176]. We then calculate the absolute percentage difference between the two means: *windowed_mean* and *global_mean*. If the percentage difference is more significant than the specified threshold (by default is 5%), then the time tick t for j^{th} VM is regarded as the change point.
- **Z-score-based Detector**: This detector is based on the calculation of the Z-scores [161, 151]. Similar to the Mean-based detector, here also a *windowed_mean*, i.e., the mean of all the values in the

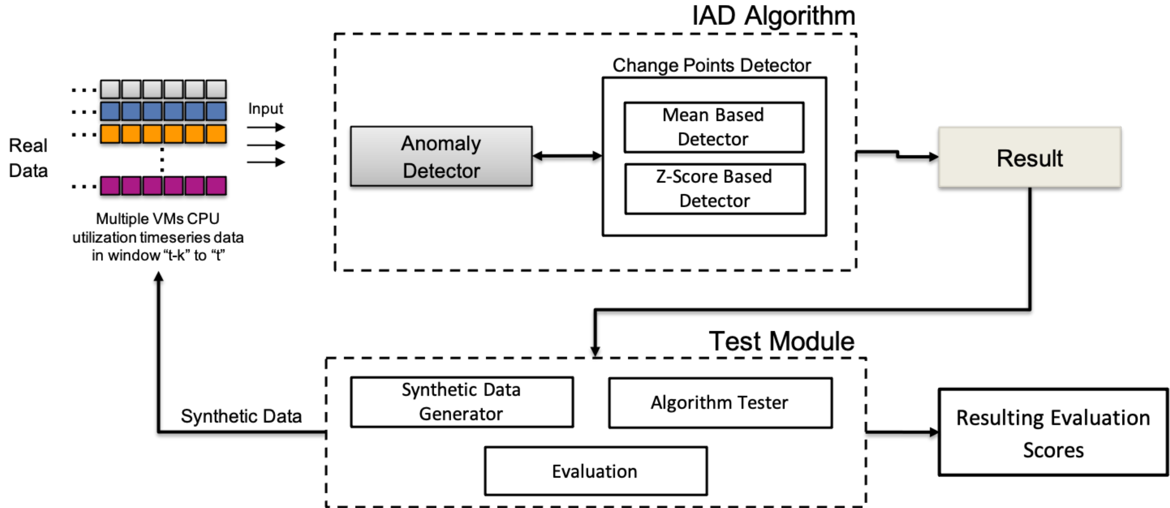


Figure 8.8.: High-level system workflow of the implemented system for evaluating IAD algorithm and the interaction between its components in a general use case.

window, and the *global_mean*, i.e., the mean of all the values until the current time tick is calculated. We additionally calculate the *global_stand_deviation*, i.e., the standard deviation of all the values until the current time tick. Since the IAD algorithm is designed for running it in an online way, *global_stand_deviation* is calculated using Welford's method [176]. These statistics are then used to calculate the z-scores for all the data points in the window using Equation 8.1.

$$z_scores = \frac{(windowed_mean - global_mean)}{\frac{global_stand_deviation}{\sqrt{w}}} \tag{8.1}$$

If the Z-scores are greater than the defined threshold ($3 \times global_stand_deviation$), then the time tick t for j^{th} VM is regarded as the change point.

In the main algorithm, only *Z-Ssore-based Detector* is used as it provides higher accuracy and has fewer false positives.

Anomaly Detector: This component receives the input resource utilization data X of size $n \times d$ where d is the number VMs hosted on a VMM along with the `minPercentVMsFault` (Table 8.3)) as the input parameter. We first check the input timeseries of w length for 1) zero-length timeseries and 2) if the input timeseries of all VMs are of the same length or not. If any of the two initial checks are true, we quit and do not proceed. We assume that all the VM's resource utilization data is of the same length only. After doing the initial checks, each of the VM's windowed timeseries belonging to the VMM is sent to the *Change Points Detector* for the detection of whether the time tick t is a change point or not. If the percentage number of VMs ($\{c_t^1, c_t^2, ..., c_t^m\}$ out of d) having the change point at time tick t is greater than the `minPercentVMsFault` input parameter, then the VMM is reported as anomalous at time tick t. The above procedure is repeated for all time ticks. Figure 8.9 shows the workflow sequence diagram of the IAD algorithm. Furthermore, the developed approach can also be applied for multiple VMMs.

8.2.2.2 Test Module

This component is responsible for generating the synthetic data and evaluating the algorithm performance by calculating the F1-score on the results from the algorithm. It consists of multiple sub-component described

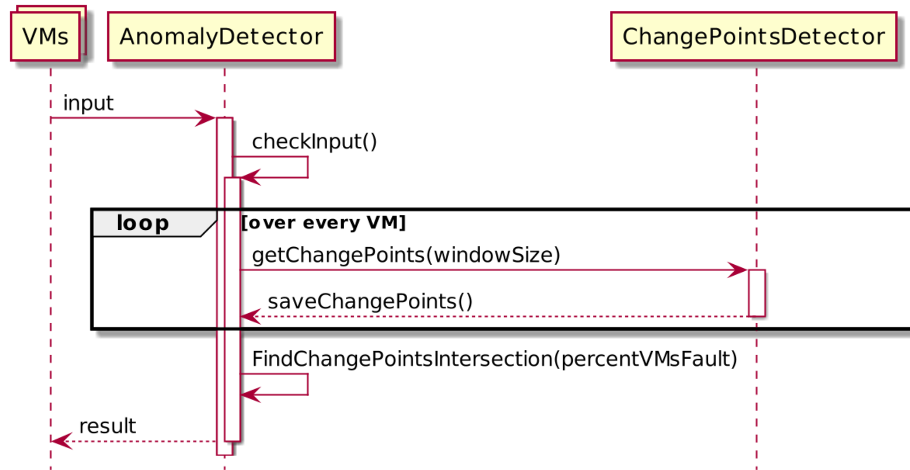


Figure 8.9.: IAD algorithm workflow sequence diagram.

below:

- **Synthetic Data Generator**: It takes the number of VMMs, number of VMs per VMM, percentage of the VMs with a fault; as the input for generating synthetic timeseries data. This synthetic data follows a Gaussian distribution based on the input parameters. This component also automatically divides the generated data into true positive and true negative labels based on the percentage of the VMs with a fault parameter.
- **Algorithm Tester**: It is responsible for invoking the algorithm with various parameters on the synthetic data and tune the algorithm's hyperparameters.
- **Evaluation**: The results from the algorithm are passed as the input to this sub-component, where the results are compared with the actual labels, and the overall algorithm score in terms of F1-score is reported.

8.2.3 Experimental Settings

We design our experiments to answer the following questions:

Q1. IAD Accuracy: How accurate is IAD in the detection of anomalous VMM when compared to other popular algorithms?

Q2. Anomalous VMMs finding efficiency and scalability: How does the algorithm scale with the increase in the data points and number of VMs?

8.2.3.1 Datasets

For evaluating the IAD algorithm, we considered four types of datasets listed in Table 8.4, and they are described below:

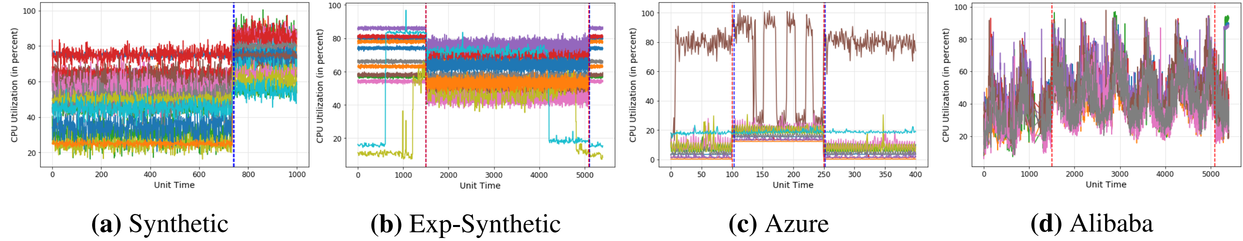
Synthetic: This is the artificially generated dataset using the *Test Module* component described in §8.2.2.

Experimental-Synthetic Merged: This is a dataset with a combination of experimental data and synthetic data. To collect the experimental dataset, we created two nested VMs on a VM in the Google Cloud Platform.

Table 8.4.: Datasets used in this work for evaluating the algorithms.

Dataset Name	Anomalous VMMs	Non-Anomalous VMMs	VMs Per VMM	TimeTicks per VM
Synthetic	5	5	10	1000
Exp-Synthetic Merged	42	17	2 (experimental) 8 (synthetic)	5400
Azure† [83]	16	10	10	5400
Alibaba† [264]	10	10	10	5400

†These are modified for our usecase.



(a) Synthetic **(b)** Exp-Synthetic **(c)** Azure **(d)** Alibaba

Figure 8.10.: An example profile of an anomalous VMM having 10 VMs in all the datasets used in this work for evaluation.

The underneath VM instance type is `n1-standard-4` with four vCPUs and `15GB` of memory, and Ubuntu `18.04` OS was installed on it. This VM instance acts as a host for the above VMs. *libvirt* toolkit is used to manage and create nested virtualization on top of the host machine. Kernel-Based Virtual Machine (KVM) is used as a VMM. The configuration of the two nested VMs are i) `2vCPU` and `2GB` memory, ii) `1vCPU` and `1GB` memory. Cloud-native web applications were run on these two VMs. Monitoring data from the two VMs and the underneath host is exported using the Prometheus agent deployed on each of them to an external VM. `stress-ng` is used for generating the load on the VMM. Based on this infrastructure, we collected a dataset for various scenarios and combined it with synthetic data.

Azure Dataset: This dataset is based on the publicly available cloud traces data from Azure [83]. We used the VMs data from it and created random groups of VMs, with each group representing the VMs hosted on a VMM. Afterward, we feed these timeseries groups in our synthetic data generator for randomly increasing or decreasing the CPU utilization of the VMs within a VMM based on the input parameters to create anomalous and non-anomalous VMMs.

Alibaba Dataset: This dataset is based on the publicly available cloud traces and metrics data from Alibaba cloud [264]. A similar method as the *Azure Dataset* was also applied to form this dataset.

Figure 8.10 shows an example profile of an anomalous VMM for all the datasets.

8.2.3.2 Evaluated Algorithms

We compare IAD to the five other algorithms listed in Table 8.5 along with their input dimension and parameters. ECP is a non-parametric-based change detection algorithm that uses the E-statistic, a non-parametric goodness-of-fit statistic, with hierarchical division and dynamic programming for finding them [140]. Branch and Border (BnB) and its online version Branch and Border Online (BnBO) are also non-parametric change detection methods that can detect multiple changes in multivariate data by separating points before and after the change using an ensemble of random partitions [131]. Lastly, we use the popular anomaly detection

Table 8.5.: The details of the algorithms used in this work for evaluation, along with their input dimension and parameters.

Algorithm	Input Dimension	Parameters
IAD	n × d	w, minPercentVMsFault
ECP [140]	n × d	change points, Min. points b/w change points
BnB [131]	n × d	w, number of trees, threshold for change points
BnBO [131]	n × d	w, number of trees, threshold for change points
IF [178]	n × d	contamination factor (requires training)
IFF [178]	n × features	contamination factor (requires training)

Table 8.6.: F1-score corresponding to each algorithm evaluated in this work (§8.2.3.2) and on all the datasets (§8.2.3.1).

Algorithm	Synthetic	Exp-Synthetic	Azure	Alibaba	Average F1-score
IAD	**0.96**	0.86	**0.96**	0.57	**0.837**
ECP	0.67	-	0.76	0.51	0.64
BnB	0.62	**0.90**	0.8	0.33	0.662
BnBO	0.87	0.81	0.86	0.4	0.735
IF	0.76	0.83	0.76	0.2	0.637
IFF	0.76	0.83	0.76	**0.66**	0.75

algorithm: isolation forest for detecting anomalous VMM [178]. The primary Isolation Forest (IF) works on the input data directly, while we also created a modified version of it called the Isolation Forest Features (IFF), which first calculates several features such as mean and standard deviation for all the values within a window on the input dataset and then apply isolation forest on it. The downside of the IF and IFF is that they require training.

8.2.3.3 Other Settings

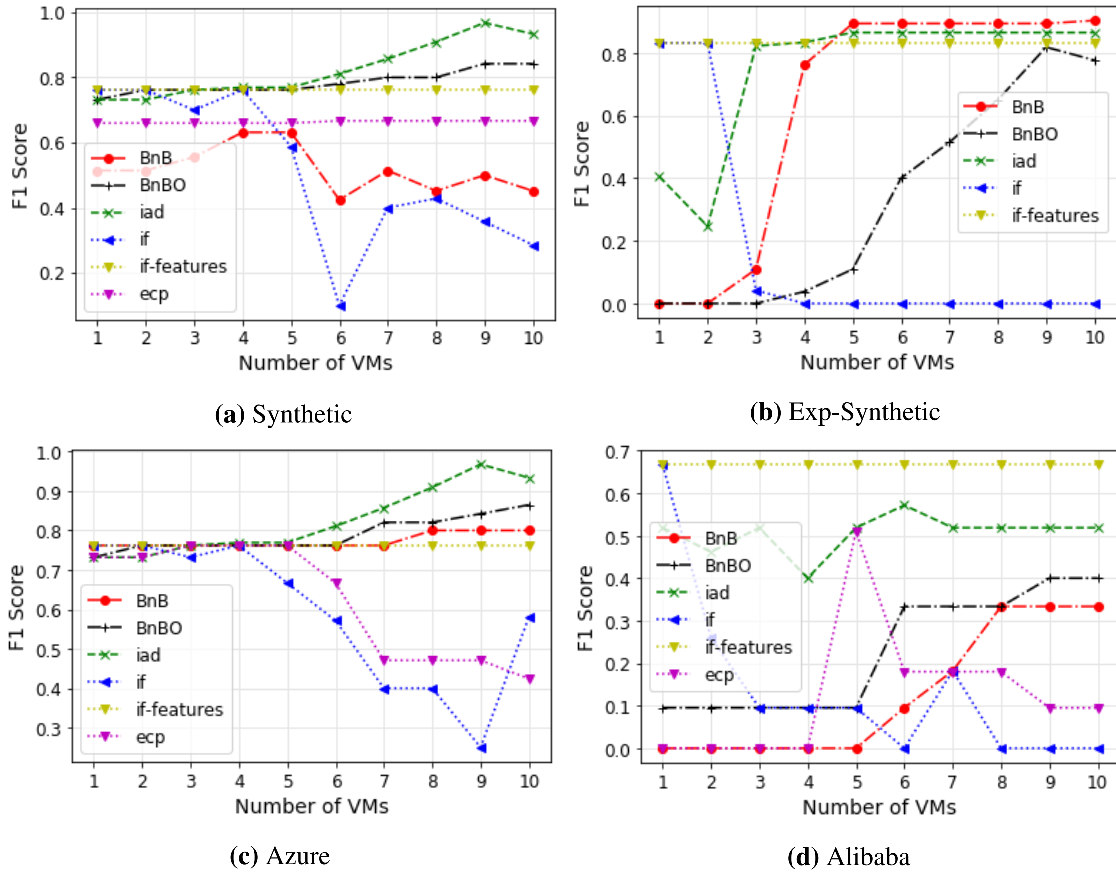
We have used F1-Score (denoted as F1) to evaluate the algorithm's performance. Evaluation tests have been executed on `2.6` GHz 6-Core Intel Core i7 MacBook Pro, `32GB` RAM running macOS BigSur version 11. We implement our method in Python. For our experiments, hyper-parameters are set as follows. The window size w is set as one minute (`60` samples, with sampling done per second), threshold k as `5%`, and `percentVMsFault` f as `90%`.

8.2.4 Results

Our initial experiments showed that 1) CPU metric is the most affected and visualized parameters in the VMs when some load is generated on the VMM; 2) All or most VMs are affected when a load is introduced on the VMM.

8.2.4.1 Q1. Indirect Anomaly Detection Accuracy

Table 8.6 shows the best F1-score corresponding to each algorithm evaluated in this work (§8.2.3.2) and on all the datasets (§8.2.3.1). We can observe that *IAD* algorithm outperforms the others on two datasets, except



(a) Synthetic **(b)** Exp-Synthetic

(c) Azure **(d)** Alibaba

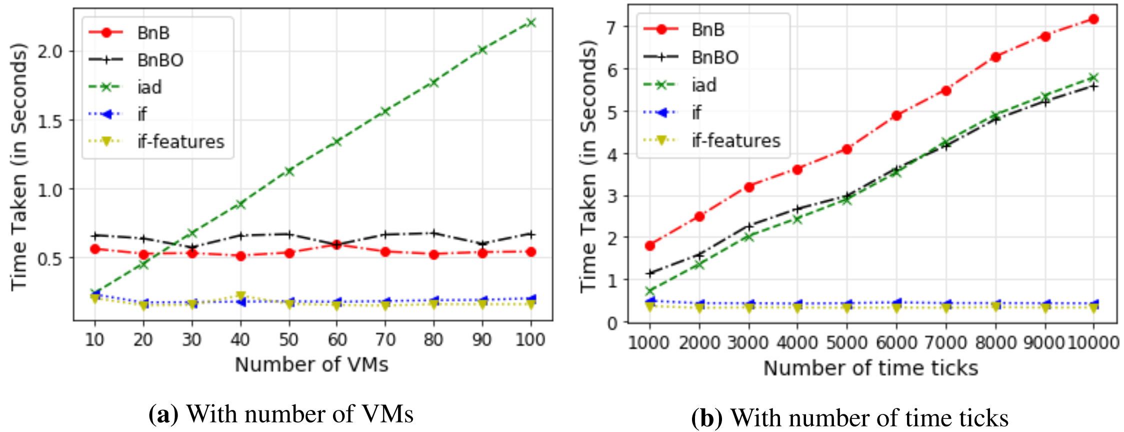
Figure 8.11.: F1-score variation with the number of VMs corresponding to each algorithm evaluated in this work (§8.2.3.2) and on all the datasets (§8.2.3.1).

for the Experiment-Synthetic dataset (BnB performed best with F1-Score of `0.90`) and Alibaba dataset (IFF performed best with F1-Score of `0.66`. However, if one wants to find an algorithm that is performing well on all the datasets (average F1-score column in Table 8.6), in that case, *IAD* algorithm outperforms all the others with an average F1-score of `0.837` across all datasets.

Furthermore, we present the detailed results of the algorithms on all four datasets varying with the number of VMs and are shown in Figure 8.11. One can observe that *IAD* performs best across all the datasets, and its accuracy increases with the increase in the number of VMs. Additionally, after a certain number of VMs, the F1-score of *IAD* becomes stable. This shows that if, for example, we have the synthetic dataset, then the best performance is possible with VMs $\geq$ `9`. Similarly, in the case of the Azure dataset, while for the Exp-Synthetic dataset, one needs at least five VMs, and for the Alibaba dataset, seven VMs for the algorithm to perform well.

8.2.4.2 Q2. Anomalous VMMs Finding Efficiency and Scalability

Next, we verify that our algorithm's detection method scale linearly and compare it against other algorithms. This experiment is performed with the synthetic dataset, since we can increase the number of VMs per VMM in it. We linearly increased the number of VMs from one to `100` and repeatedly duplicated our dataset in time ticks by adding Gaussian noise. Figure 8.12 shows various algorithm's detection method scalability



(a) With number of VMs

(b) With number of time ticks

Figure 8.12.: Algorithm's detection method scalability with respect to different parameters.

for different parameters. One can observe that *IAD's* detection method scale linearly in terms of both the parameters. However, when the number of VMs are scaled to `100`, *IAD* takes a longer time as compared to others, but it provides results under `2.5s` which if we see is not that much considering the accuracy we get with that algorithm. However, on the time ticks parameter, *BnB*, *BnBO* and *IAD* performed similar to each other, while *IF* and *IFF* provides results under `1` second, but its accuracy is worse as compared to the others on all the datasets, and it has the extra overhead of training. *ECP* algorithm's results are not shown, since it requires more than an hour for performing the detection with `100` VMs and `100,000` time ticks.

8.3 Summary

In this chapter, we described two anomaly detection algorithms for FDN: 1) Online memory leak detection in VMs using *Precog* in §8.1, and 2) Anomalous VMMs detection using *IAD: Indirect Anomaly Detection* in §8.2.

Memory leak detection has been a research topic for more than a decade. Many approaches have been proposed to detect memory leaks, most of them looking at the application's internals or the object's allocation and deallocation. The *Precog* algorithm is most relevant for the serverless compute clusters, where the cloud administrator does not have access to the source code or know about the internals of the deployed applications. The performance evaluation showed that the *Precog* could achieve a F1-Score of `0.85` with less than half a second prediction time on the real workloads.

The proposed *IAD* algorithm is useful for detecting anomalous VMMs in serverless compute clusters. We compared it against the popular change detection algorithms, which could also be applied to the problem. *IAD* algorithm outperforms all the others on an average across four datasets by `11%` with an average accuracy score of `83.7%`. *IAD* algorithm scales linearly with the number of VMs hosted on a VMM and the number of time ticks. It takes less than `2.5` seconds for *IAD* algorithm to analyze `100` VMs hosted on a VMM for detecting if that VMM is anomalous or not. This allows it to be easily usable in the cloud environment where the fault-detection time requirement is low and can quickly help DevOps to know whether the problem is of the hypervisor or not.

9

Function Delivery Network Evaluation Settings

“What you get by achieving your goals is not as important as what you become by achieving your goals."

— Zig Ziglar

After describing the broad FDN framework, we, in this chapter, explain the methodology used to carry out its performance evaluation. The performance evaluation aims to understand the general performance of FDN in diverse scenarios. To investigate the performance, we first introduce the different benchmarks, i.e., FaaS functions, along with the developed application we use to evaluate in §9.1. Following this, we describe the different heterogeneous clusters used in this work to form the edge-cloud continuum within FDN in §9.2. Lastly, in §9.3, we describe the complete evaluation infrastructure and the different performance quality metrics used for the evaluation.

9.1 Benchmarks

When a function microbenchmark is executed on any computing system within a cluster in FDN, it will require a set of resources to operate correctly. If not enough resources are provided, its execution may slow down. Moreover, if a system is overloaded with processes that devour its resources, any running program will be affected, and the overall performance will fall. This situation will likely happen on the edge clusters when too many functions are executed simultaneously. Thus to understand the behavior of FDN under different situations, we have identified a subset of the microbenchmarks provided with the FaaSProfiler [263]. We have put them under different categories based on their use cases and the system resource they need the most (§9.1.1). In order to evaluate FDN in a real-world scenario, we created an application with these microbenchmarks described in §9.1.2.

9.1.1 FaaS Functions

We have identified five main categories under which we have placed the functions to be evaluated. While the implementation logic of the used functions is not relevant, the goal is to simulate the workload that would occur in a real scenario, even though the deployed functions may not be used in reality. The functions are also summarized in Table 9.1 along with their category, description, and language runtimes.

9.1.1.1 Web-based FaaS Functions

This category simulates a typical web-based workload that does not require heavy computation. We have considered the following function in this category for the evaluation:

nodeinfo: This function returns the basic characteristics of the node on which it is running, like CPU count, architecture, and uptime. It is used to test the environment under soft load since it does not require any heavy computation or resources, and therefore, it should take the same execution time as any setting.

9.1.1.2 CPU-Intensive FaaS Functions

CPU-intensive functions are one of the most popular types of programs nowadays. Such functions require mainly CPU power and a small amount of memory and network resources [232]. We have considered the following three functions under this category:

primes: It takes as input an integer `n` and computes the number of primes lower than `n`. Its goal is to force a high load on the CPU. The algorithm used in this function is a standard one that we have modified with the possibility of specifying the threshold. In this way, we could tune the function's performance to the clusters' characteristics. In the experiments, we have set `n = 1000`.

linpack: It solves a dense linear system of equations in double precision and returns the results in GFlops. Problem size (number of equations) is fixed to `100`.

sentiment-analysis: It analyzes the sentiment of a provided string using the Python TextBlob library.

9.1.1.3 Memory-Intensive and Disk I/O-Intensive FaaS Functions

The storage disk and the memory are the target resources in this category. These resources will be responsible for influencing the execution times of the function. We have considered the following two functions:

gzip-compression: This function takes as input an integer *file_size*, writes *file_size* GB on a file on the disk and then read the same amount in the memory. The returned value is the time taken to perform the two operations.

dd: This function utilizes the Unix command-line utility dd - that stands for "Data Definition" [177]. Its main usage is copying files from different locations, with the advantage that it is more efficient than other commands when dealing with large documents such as device files. For our purposes, we copy */dev/zero* - which produces a continuous stream of zero value bytes - into a temporary directory */tmp/out*, so that we do not mess up the cluster's disk. Consequently, this function significantly stresses the disk usage like the *gzip-compression* function.

Table 9.1.: Summary of the FaaS functions microbenchmarks used as part of this work for evaluating FDN.

Category	Num	Microbenchmark	Description	Runtime
Web	Fn1	nodeinfo	Gives basic characteristics of node like CPU count, architecture, uptime	Node.js 14
CPU	Fn2	primes-python	Calculates prime numbers till 1000.	Python 3.7
	Fn3	linpack	It solves a dense linear system of equations in double precision and returns the results in GFlops. Problem size (number of equations) is fixed to 100.	Python 3.7
	Fn4	sentiment analysis	Analyzes the sentiment of a provided string using the Python TextBlob library	Python 3.7
Mem & Disk	Fn5	dd	It is based on Unix dd command-line utility for converting and copying files. 128bytes as block size and five times conversion is used as parameters.	Python 3.7
	Fn6	gzip-compression	Creates a file with random numbers of size 1MB and compresses it using gzip compression.	Python 3.7
Network	Fn7	json-loads	It fetches a big JSON file from the internet, loads it in memory, and converts it into a string.	Python 3.7
ML-based	Fn8	ml-lr-prediction	It first downloads a linear regression model trained on user reviews data from the storage bucket along with the test data and performs prediction on it.	Python 3.7
	Fn9	image-processing	It reads an image from object storage (here MinIO) and performs basic operations (flip, rotate, filter, grayscale and resize) on the image.	Python 3.7

9.1.1.4 Network I/O-Intensive FaaS Functions

json-loads: As the name suggests, this function sends a request to a remote URL containing a JSON file and waits for the response. Then, it loads the file, converts it into a string, and returns the time needed to perform all the operations. The resource that this function utilizes is the network I/O.

9.1.1.5 ML-based FaaS Functions

lr-prediction: This function first downloads a linear regression model trained on user reviews data and the test data from the storage buckets located on the GCP. It then performs prediction using the downloaded model on the test data. The returned value is the time taken to perform the prediction.

image-processing: It takes the image name and the object storage (here MinIO) credentials as input. Based on it, it downloads the image from the object storage and performs basic image operations flip, rotate, filter, grayscale, and resize on the image. The returned value is the time taken to perform the operations.

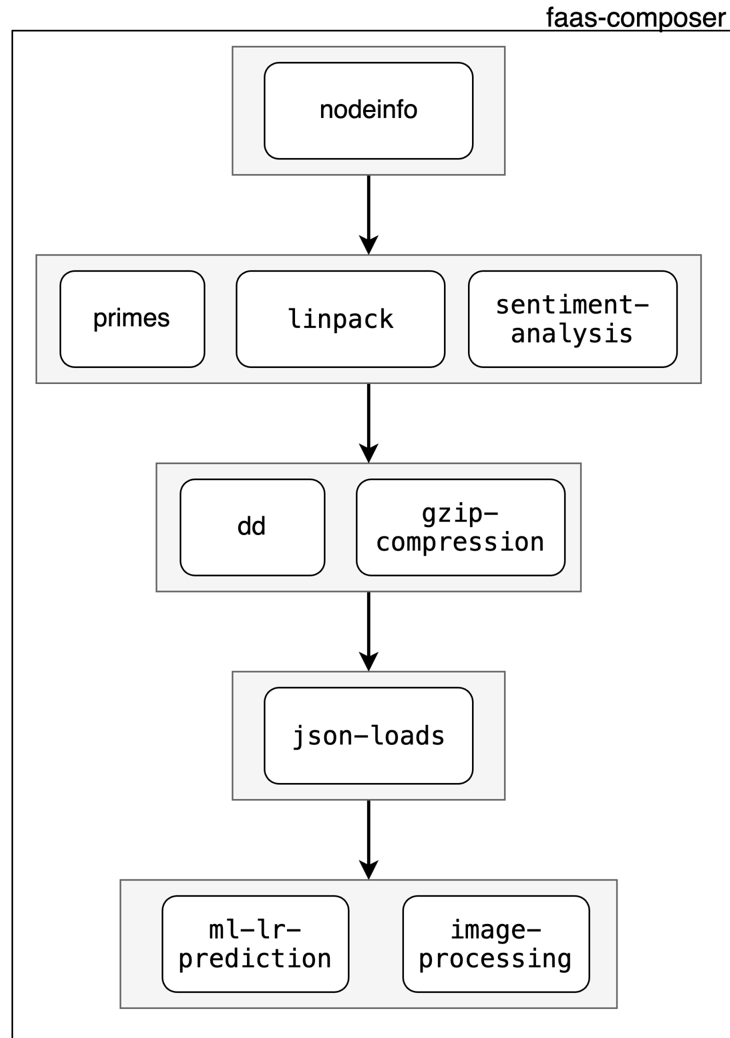
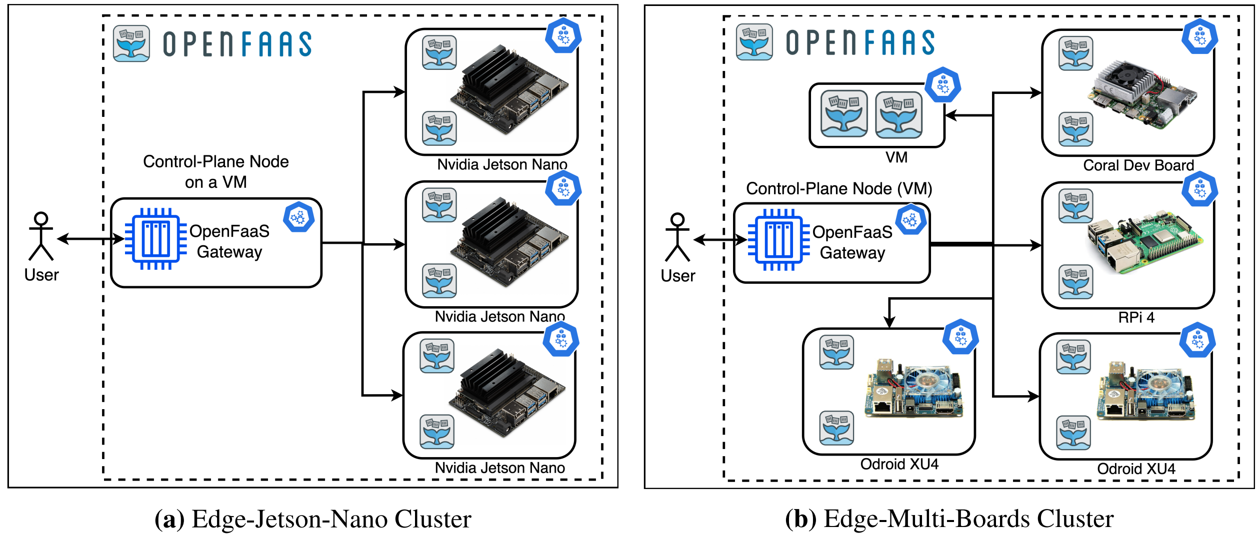


Figure 9.1.: High level workflow of the application used in this work for evaluation.

9.1.2 Serverless Application

We created an application called `faas-composer`, with the microbenchmarks described in §9.1.1. The overall workflow of the application is shown in Figure 9.1. The functions in the gray boxes are executed in parallel, while these are executed in sequence. The application flow starts with the `node info` function, which exposes an HTTP endpoint and provides the user with basic information about the system, such as hostname, underlying architecture, and number of CPUs. After it returns the response, `faas-composer` invokes all the compute-intensive functions: `primes-python`, `unpack` and `sentiment-analysis` asynchronously. The `faas-composer` waits for their responses to come back. Once the responses are received, then the `dd` and `gzip-compression` functions are executed asynchronously. The `faas-composer` waits for their responses, and then it invokes `json-loads`. After receiving the response from it, `faas-composer` invokes `ml-lr-prediction` and `image-processing` functions asynchronously. `ml-lr-prediction` queries the model and data from the Google cloud storage (created in GCP in the `Europe-west3` region) and then performs prediction. Once the responses are available from both invocations, `faas-composer` sends back the overall execution time it took to invoke all the functions back to the user.



(a) Edge-Jetson-Nano Cluster

(b) Edge-Multi-Boards Cluster

Figure 9.2.: Schematic high-level diagrams of the two edge clusters based on the embedded devices with limited resources used in this work for FDN's evaluation.

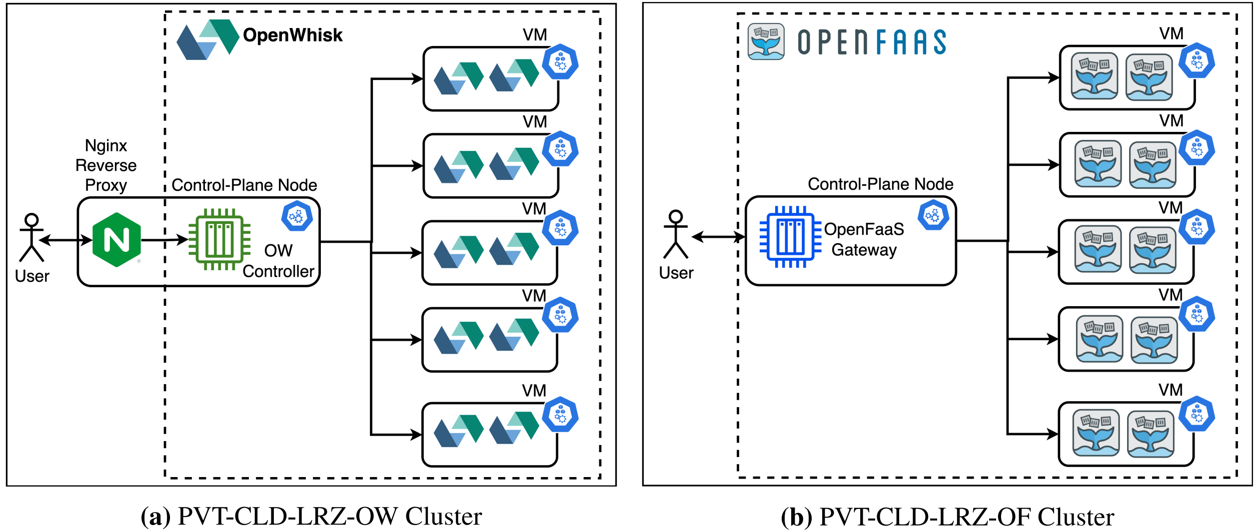
9.2 Heterogeneous Target Serverless Compute Clusters

The clusters are the operative component of the FDN and contain the deployed functions. A single cluster is a group of servers on top of which a serverless compute platform is deployed, responsible for managing and executing the functions. We targeted multiple heterogeneous serverless clusters to demonstrate the functionality of the FDN. From the point of view of the FDN's courier control plane, the clusters are blocks, that it can attach to and that can be removed or replaced at any time as long as any action is notified to the control plane. The employed clusters spread across the edge-cloud continuum are described in the following subsections. These clusters are created based on the automation approach mentioned in §4.2.1.2. Also, the configuration of each target cluster, the serverless compute platform used, and the number of nodes present in that target cluster are summarized in Table 9.2.

9.2.1 Edge-Clusters

The edge-clusters are based on embedded devices with limited resources. In order to not overload the edge devices, we created the Kubernetes control-plane on a VM, and edge devices join as worker nodes to it. The monitoring solution and Kubernetes control-plane components are only scheduled to run on the VM. It allows keeping the edge devices free for functions and harnesses their whole compute power. It is to be noted that the VM is based on the AMD64 architecture while the edge devices are based on ARM CPU architecture; therefore, we have to build Kubernetes and the monitoring solution container images for multi-architecture. Furthermore, the deployment of the functions only on edge devices was controlled by making sure the *OpenFaaS-Fn* namespace (in which OpenFaaS schedule all the functions) works only on edge devices. We have created two edge clusters for the evaluation of FDN described as follows:

Edge-Jetson-Nano: This edge cluster consists of three embedded Nvidia Jetson Nano devices [288]. OpenFaaS support low-end devices and provides binaries for ARM processors; therefore, we utilized OpenFaaS on top of k3s [243], a lightweight version of Kubernetes, to host a Kubernetes cluster. Additionally, a monitoring solution based on Prometheus is deployed within this cluster for *FDN-Monitor* (§4.2.2) to gather various metrics. Figure 9.2a shows the high-level diagram of this cluster. The functions created with OpenFaaS



(a) PVT-CLD-LRZ-OW Cluster

(b) PVT-CLD-LRZ-OF Cluster

Figure 9.3.: Schematic high-level diagrams of the two private cloud clusters based on two serverless compute platforms used in this work for the FDN's evaluation.

has by default the invocation URL pattern as `<server-ip>/function/<function-name>` pattern, which is the same required by the *FDN's Courier Load Balancer* (§6.2) for invocation. Therefore, we do not have to deploy reverse proxy in the clusters based on OpenFaaS.

Edge-Multi-Boards: This second edge cluster consists of five heterogeneous embedded devices: Google Coral Dev board, Nvidia Jetson Nano, Raspberry PI4, and two Odroid XU4 devices. We again utilized OpenFaaS on top of k3s [243] to host a Kubernetes cluster on this cluster. A monitoring solution based on Prometheus is also deployed within this cluster for *FDN-Monitor* (§4.2.2) to gather various metrics. We also added another VM in this cluster as a worker node to share the control-plane workloads. Figure 9.2b shows the high-level diagram of this cluster.

9.2.2 Cloud-Clusters

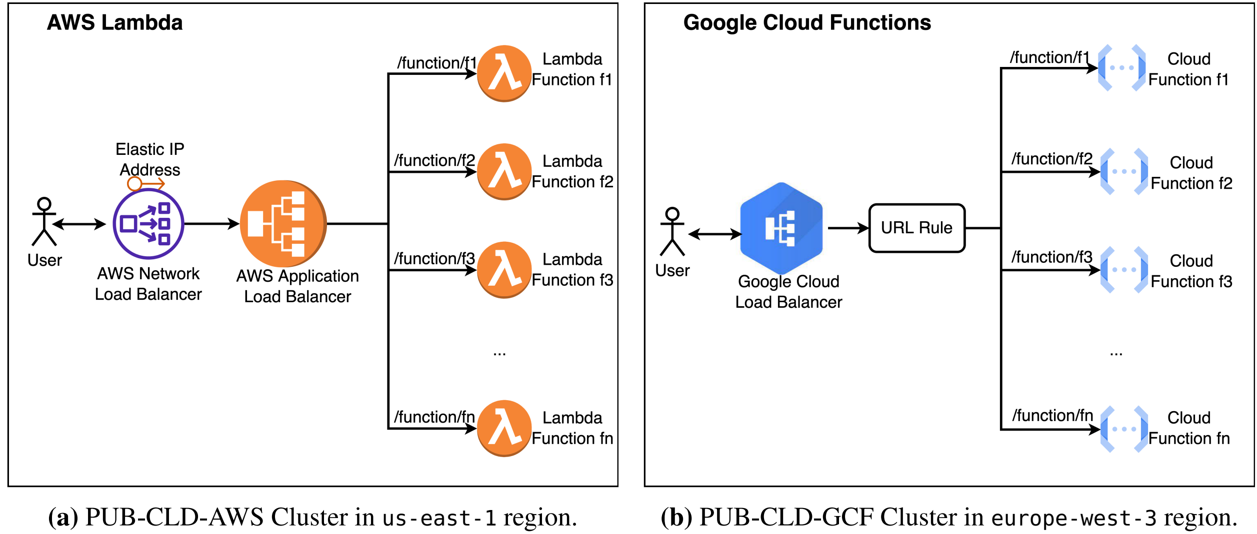
Cloud clusters represent the clusters in the cloud. These clusters could be hosted on a private or public cloud. We have employed both types of clusters, presented in the following subsections.

9.2.2.1 Private-Cloud-Clusters

This cluster type is hosted on the VMs created on-premises private cloud. This cluster type is essential for running functions that work on private data residing on-premise. We created two of these clusters:

PVT-CLD-LRZ-OF: This private *cloud-cluster* is composed of five VMs hosted on a private cloud at the LRZ [182]. LRZ cloud is based on OpenStack. Each VM has four vCPU cores and `16GiB` of memory. We first deploy a fully-fledged Kubernetes cluster on the VMs using *kubeadm*. Then we deploy OpenFaaS serverless compute platform on it. This cluster is also deployed with a Prometheus-based monitoring solution. A high-level cluster architecture diagram is shown in Figure 9.3b.

PVT-CLD-LRZ-OW: This private *cloud-cluster* is also composed of five VMs on a private cloud at the LRZ. Each VM has four vCPU cores and `16GiB` of memory. We first deploy a fully-fledged Kubernetes



(a) PUB-CLD-AWS Cluster in `us-east-1` region. **(b)** PUB-CLD-GCF Cluster in `europe-west-3` region.

Figure 9.4.: Schematic high-level diagrams of the two public cloud clusters used in this work for FDN's evaluation.

cluster on the VMs using *kubeadm*. Then we deploy OpenWhisk as a serverless compute platform on it. The functions deployed on the OpenWhisk cluster require authentication [226]. Therefore, we have also deployed a reverse proxy based on NGINX [208] in the cluster, containing proper authorization. The reverse proxy attaches the authorization to each incoming function's request, making it easier to invoke from FDN. Additionally, for keeping the function invocation URL in `<server-ip>/function/<function-name>` pattern, we created a URL rule within the reverse proxy to map the invocation URLs (`.../function/*`) with the OpenWhisk function URLs. This cluster is also deployed with a Prometheus-based monitoring solution. A high-level cluster architecture diagram is shown in Figure 9.3a.

9.2.2.2 Public-Cloud-Clusters

These clusters are created using the public cloud serverless compute platforms. We have considered the following clusters for this type:

PUB-CLD-AWS: This public *cloud-cluster* is created using the AWS Lambda serverless compute platform. A high-level cluster architecture diagram is shown in Figure 9.4a. This cluster is created in the `us-east-1` region. The Lambda functions created on AWS can be assigned with the *Function URL* service from AWS, but each function's URL prefixes are assigned randomly [68]. To keep the prefix for each function URL constant, we created an *AWS Application Load Balancer* [32] and used it as the trigger entry point for each function. Furthermore, since FDN's *Courier Load Balancer* (§6.2) requires the function invocation URL to be in `<server-ip>/function/<function-name>` pattern, we created rules within the AWS Application Load Balancer to map the invocation URLs (`.../function/<function-name>`) with the desired Lambda functions [27]. *FDN's Courier Load Balancer* (§6.2) requires the cluster to have a static IP address for load balancing across multiple clusters. It cannot use the domain name for load balancing. Thus, we created an *Elastic IP* address in AWS. However, one cannot assign the *Elastic IP* address to an *AWS Application Load Balancer* [26]. Consequently, we created an AWS Network Load Balancer [31] on top of the *AWS Application Load Balancer* [21] and assigned the *Elastic IP* address to it [25]. We could also use the *AWS Network Load Balancer* as a trigger for the Lambda functions, but we cannot create the URL mapping required by the FDN within it. Therefore, we have to keep the *AWS Application Load Balancer*. The final

Table 9.2.: Different target heterogeneous clusters spread across edge-cloud continuum used for evaluating the FDN.

Category	Cluster Name	Device/VM	Platform	Region	Nodes
Edge	Edge-Jetson-Nano	Nvidia Jetson Nano	OpenFaaS	europe-west-3	3
	Edge-Multi-Boards	Coral Dev board, Jetson Nano, RPI4, two Odroid XU4	OpenFaaS	europe-west-3	4
Private Cloud	PVT-CLD-LRZ-OF	VM, 4 vCPU 16GiB	OpenFaaS	europe-west-3	5
	PVT-CLD-LRZ-OW	VM, 4 vCPU 16GiB	OpenWhisk	europe-west-3	5
Public Cloud†	PUB-CLD-GCF†	N/A†	GCF	europe-west-3	N/A†
	PUB-CLD-AWS†	N/A†	AWS Lambda	us-east-1	N/A†

† Host VMs or containers configuration information in which functions are deployed is not available.

cluster architecture diagram is shown in Figure 9.4a. We use the *AWS Cloud Watch* [81] to extract the function's performance metrics.

PUB-CLD-GCF: This public *cloud-cluster* is created using GCF in the `europe-west-3` region. The functions created in GCF are by default assigned a URL based on the project-id in which those are created. These URL prefixes are unique since all the functions are created within the same project. However, as FDN's *Courier Load Balancer* (§6.2) requires the cluster to have a static IP address for load balancing, we created a *Google Cloud Load balancer* [235], which, by default, has a public IP address. Furthermore, for keeping the function invocation URL in <server-ip>/function/<function-name> pattern, we created URL rules within the *Google Cloud Load balancer* to map the invocation URLs (`.../function/*`) with the functions having the names as the last part of the URLs. We use the *Google Cloud Monitoring* [117] solution to extract the function's performance metrics. A high-level architecture diagram for this cluster is shown in Figure 9.4b.

9.3 Evaluation Infrastructure

This section presents the evaluation infrastructure used in this work for assessing FDN. We start with the FDN deployment settings in §9.3.1. In §9.3.2, we present an evaluation framework, which is a framework around the FDN architecture. Its purpose is to test the FDN under different scenarios by replicating user workload patterns, collecting various metrics data, and plotting the graphs.

9.3.1 FDN Deployment Settings

We deployed each component of FDN (§4.2) in a Kubernetes cluster consisting of three VMs. Each VM has Ubuntu `18.04`, `2.4` GHz xeon skylake processor, `4`vCPU cores, and `8GB` memory. Additionally, we mount a common Network File System (NFS) storage point on each VM to keep the data consistent across VMs and allow easier scalability. Each component of FDN is deployed as a `replica-set` workload within the Kubernetes cluster. Each `replica-set` has one replica and is attached with a Persistent Volume Claim (PVC) for storing all the data related to the component. A PVC is a request for storage by a user for the

pod, and PVCs consume Persistent Volume (PV) resources. A PV is a storage piece in the cluster that an administrator has provisioned.

The weights of the clusters within *Courier Load Balancer* can only be dynamically updated by updating them inside a Linux socket file (`/var/run/hapee-lb.sock`). *Courier Control Plane* needs to access it for dynamically updating the weights with no downtime. Therefore, we deployed the *Courier Load Balancer* and the *Courier Control Plane* components in different containers but within the same pod with the shared namespace. Sharing the namespace allows the *Courier Control Plane* to access the Linux socket file and dynamically update the weights. Each `behavioral modeling` mode and *Courier Control Plane*'s load balancing algorithms are deployed on demand as a pod within the cluster based on the requirement and the set load balancing algorithm. These deployed components are specific for each function and scale up or down with the increase or decrease in the number of functions. Each *Virtual Kubelet* component is also created as the `replica-set` within the same cluster. All the FDN related components are deployed in the `fdn-related-stuff` namespace within the Kubernetes cluster. Figure 9.5 shows the high-level overview of the various FDN components deployed in the Kubernetes cluster.

When we deploy a function on a serverless compute cluster, a corresponding pod is created within the FDN cluster. The pod is assigned to the *Virtual Kubelet* node representing the cluster, using the node selector parameter. We need to create pods with the same name in the FDN cluster assigned to their corresponding *Virtual Kubelet* nodes for deploying the same function on the other clusters. However, Kubernetes does not allow running the pod with the same name in the same namespace. One alternative could be to change the function name and append the cluster name to it. However, we wanted to keep the function name consistent across clusters. Therefore, we created namespaces for each *Virtual Kubelet* node. The pod created for the corresponding function for a particular cluster goes to the namespace belonging to that cluster. It allows us to keep all the pods related to a cluster within a namespace, and we can now create multiple pods with the same name representing the function deployed in different clusters.

9.3.2 FDN Test Framework

In order to test FDN under the different scenarios, we created an extra framework around it. It replicates the user workload patterns, collects various metrics data, and plots the graphs. A high level workflow diagram of the *FDN Test Framework* is shown in Figure 9.6. A configuration file containing all the necessary parameters is passed as input to the framework (step ❶). This file defines all the scenarios that the framework will execute. For each scenario listed in the configuration file, the client configures the FDN, such as deploying the function to the cluster(s) and configuring the load balancing strategy (step ❷ - ❸). Once the FDN is configured, a load generator instance is initialized (step ❸). We use *k6* as the load generation tool [306]. This load generation instance replicates the past user requests pattern (step ❺) and sends them to either the FDN's *Courier Load Balancer* endpoint or the deployed function's endpoint on a particular cluster based on the evaluation scenario (step ❻). These requests patterns are the daily accesses to some Wikipedia pages, representing the times a webpage is accessed in a day over the years. The dataset is accessed from Kaggle [310]. Once the test is over, the framework collects all the metrics data stored in the FDN *Monitoring database* and saves them as local CSV files for graph plotting (step ❼-❽). The user can check the FDN's *Grafana* to see the graphs of different metrics in real time during load generation. Once a scenario is finished, the client sleeps for five minutes. This waiting time is needed to reset the clusters so that a new scenario is not influenced by the last one.

The *FDN Test Framework* runs on a different Kubernetes cluster than the one where the FDN is located. It is necessary to avoid any interference and resource blockages by load generation on the FDN components. We deployed it on a Kubernetes cluster consisting of two VMs. Each VM has Ubuntu `18.04`, `2.4` GHz xeon

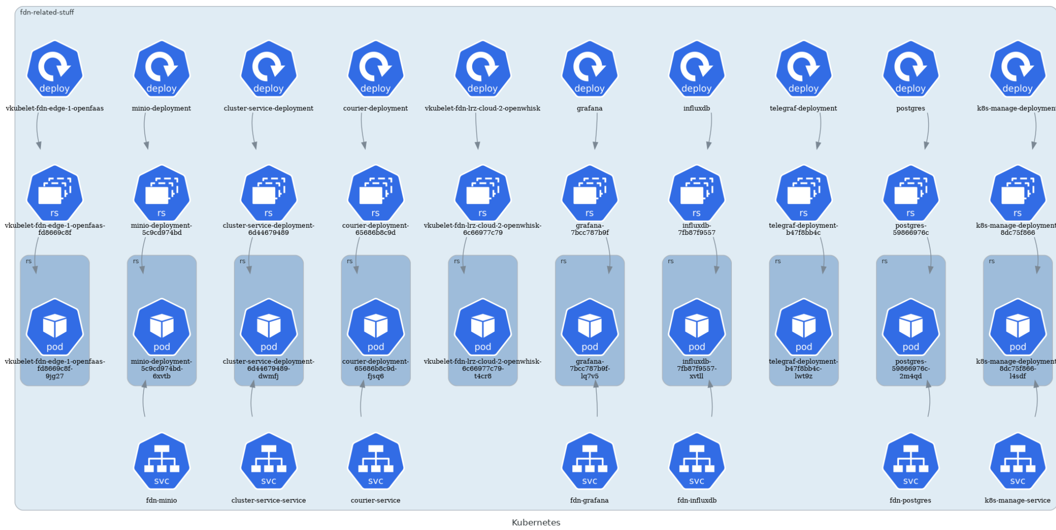


Figure 9.5.: A high-level overview of the various FDN components when deployed on the Kubernetes cluster within the `fdn-related-stuff` namespace.

skylake processor, `4`vCPU cores, and `4GB` Memory. In the following subsections, we briefly explain three components of the *FDN Test Framework*.

9.3.2.1 Configuration File

The configuration file is a YAML file supplied as input to the framework. Here all the scenarios on which FDN needs to be tested are present. It also contains the configuration parameters pertaining to the FDN. Its content can be seen in Listing A.3 in Appendix A.

9.3.2.2 FDN Test Client

The role of the client is simply parsing the input YAML configuration file and acting accordingly. Based on the parsed file, it is responsible for running the scenarios. It first configures the FDN for each scenario with tasks such as deploying or undeploying the function and setting the load balancing algorithm. Once the FDN is configured, it instructs *FDN Load Generator* to run the load generation on the configured endpoint for the desired time and invocation trace pattern. After the load generation is complete, it collects all metrics data from the FDN *Monitoring database*, save it as local CSV files, and plot graphs from them.

9.3.2.3 FDN Load Generator

As we already have explained in the flow description of *FDN Test Framework*, a *k6* instance is used to simulate the requests flow. *k6* is an open-source load testing tool providing the best developer experience

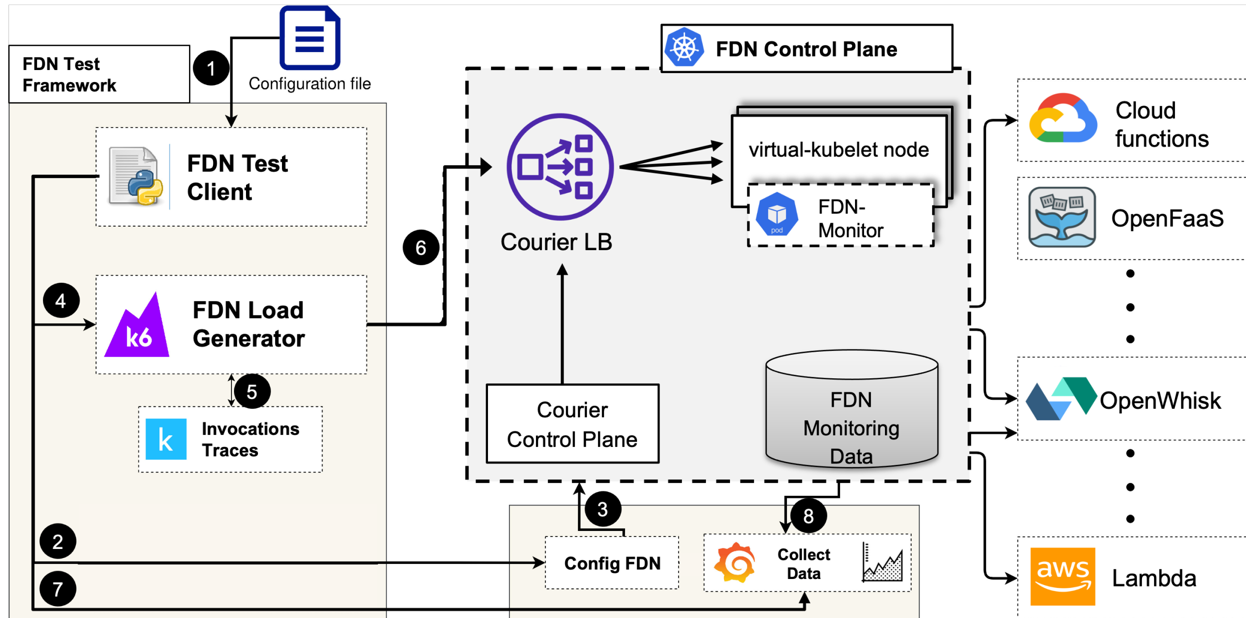


Figure 9.6.: Workflow of the FDN Test Framework. Its purpose is to test FDN under different scenarios. The configuration file is the input to the framework, containing all the scenarios to execute. The performance metrics data for the scenarios and graphs are the general output of the framework. The client within the framework starts the execution of the load generation by simulating the user workload patterns using the *k6* tool.

for API performance testing [306]. *k6* offers the possibility to create multiple Virtual Users (VUs) that simulate the requests of real users. In order to start a *k6* instance, it is necessary to define an execution script that specifies the general settings used and the job each VU must perform. This job is simply the request (or multiple requests) that the VU is meant to execute, and once it is accomplished, the VU terminates. In our case, the routine is the invocation of one of the functions deployed directly on the target cluster or through the *Courier Load Balancer* endpoint. One of the advantages of *k6* is that we can decide the rate used by VUs to generate requests. It is necessary to replicate a real access pattern, since having a single user would not create meaningful results. For this reason, we define a sequence of stages where the number of VUs ramp up and down to reach the desired number. For each stage, the number of VUs are decided based on the past user workload traces described below.

Invocations Traces: The way the requests arrive at the *Courier Control Plane* within FDN is also an important aspect that must be considered. Even though the user client is not an entity controlled by us, and we do not know how it will behave, we need to replicate a real-world scenario and provide a proper flow as similar as possible to an actual requests sequence. In order to do so, we replicate past user request patterns called *Invocations Traces* in this work. These *Invocations Traces* are the daily accesses to some Wikipedia pages, representing the times a webpage is accessed in a day over the years. The dataset is accessed from Kaggle [310]. An extract of this dataset can be seen in Table 9.3. Each line represents a Wikipedia page, while the columns correspond to the date of the period from 2015 to 2019. Therefore, each cell contains the number of visitors for the given page on the given date. In order to use this data, it was necessary to perform two adjustments. First, each day corresponds to a period of `10` seconds in our tests. In this time frame, our load generation instance would ramp up or down the number of VUs to reach the desired number. The second adjustment involves the actual number of VUs. In a real scenario, the number of visitors to a Wikipedia page may grow to thousands. We cannot satisfy so many requests; therefore, we scaled down the

Page	2015-07-01	2015-07-02	2015-07-03	2015-07-04	...
The_Avengers_(2012_film)_en	3698	3470	3519	4057	...
Avengers:_Infinity_War_en	54	59	40	46	...
Bayern_Munich_fr.wikipedia	338	280	261	300	...
Interstellar_de	6	5	2	8	...

Table 9.3.: An extract of the dataset containing the number of visits to four Wikipedia pages on different dates. Each row represents a Wikipedia page, while the columns correspond to the date of the period from 2015 to 2019. Each cell contains the number of visitors for the given page on the date. In our tests, each day corresponds to a period of `10` seconds. The number of visitors is used as the number of function invocations. However, we scaled down the number of invocations so that the maximum number of VUs is less than `200`.

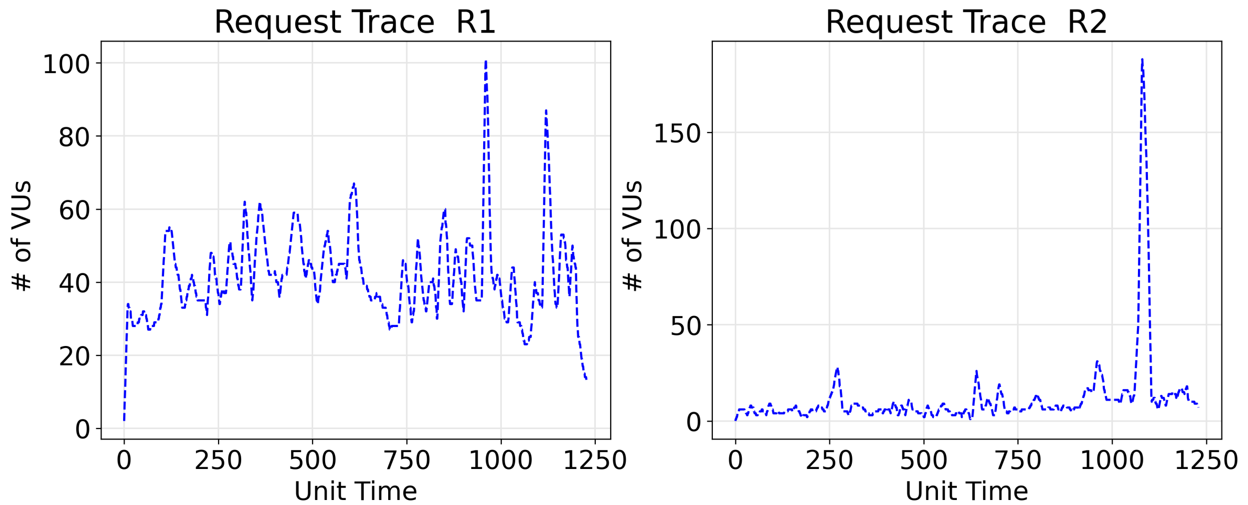


Figure 9.7.: Visualization of the function invocation traces in terms of VUs (y-axis) used in this work for FDN evaluation. The x-axis represents the unit time, where one unit time represents `10` seconds.

numbers so that the maximum number of VUs is `200`. For our evaluation, we have selected two *Invocations Traces*, and their visualization can be seen in Figure 9.7:

1. **R1 High-workload**: The first function *Invocations Trace* presents a pattern with a continuous high load, with the number of VUs that varies between `30` and `70`. Nonetheless, the VUs are not constant but sharply increase and decrease, forming a sequence of copious spikes.
2. **R2 Low-workload**: The second function *Invocations Trace* represents a low load, with all the VUs below `25` for the entire pattern length. The only exception is a high spike that reaches the value of almost `200` towards the end of the experiment.

9.4 Performance Quality Metrics

In our evaluations, the total test duration for each scenario is fixed to `20 minutes` using the two *Invocations Traces* (R1 and R2, §9.3.2.3). The total duration for which the metrics data is collected is set to `30 minutes` and the sampling rate is set to `60 seconds`, i.e, metrics values are aggregated for `60` seconds.

For analyzing the results, we have mainly used the following metrics, classified into two categories:

9.4.1 User-Centric Metrics

We use the following user-centric metrics to compare the performance of functions and load balancing algorithms:

1. **Response time**: It represents the execution duration of a request sent by the user. We use its two aggregations: average and 90th percentile (P90). The response time for an HTTP request below which 90% of the response time values lie is called the 90th percentile (P90) response time, which means 90% of the requests are processed in P90 response time or less. This metric is vital from the SLO point of view, where one wants to have most of the requests (90% in this case) completed before a specific time. This metric is seen by the client, helping to overall judge the performance.

2. **Mean Execution Time (MET)**: The time a function code spends processing an event is called execution time. This metric represents the mean of execution times for all the successful invocations that happened within the evaluation period. The mathematical form is given by Equation 9.1. This helps to rank the cluster based on their performance (the lower the number, the better the cluster).

$$MET(f) = \frac{\sum_{i=1}^{n} \text{execution_time}_i}{n} \tag{9.1}$$

   where n is the total number of successful invocations.

3. **Mean Successful Invocations per minute (MSI/min)**: If the function code is successfully executed on a serverless compute platform, it is called a successful invocation. This metric represents the mean of all the successful invocations per minute from the total invocations during the evaluation. The mathematical form is given by Equation 9.2. It helps us to know the capabilities of the clusters.

$$MSI/min(f) = \frac{\sum_{i=1}^{m} \text{successful_invocations}_i}{m} \tag{9.2}$$

   where i represents the minute i and m is the total number of minutes the evaluation is conducted invocations.

9.4.2 Platform-Centric Metrics

We use the following metrics to compare the resource usage of functions on various clusters. We use two aggregations for these metrics: 1) average aggregation per minute and 2) average aggregation during the entire duration of the test.

1. **Instances**: It represents the number of active concurrent *Function Instance* (container or MicroVMs) for serving the user invocations on a serverless compute platform (§2.3.1). Platforms increase or decrease the number of *Instances*, depending on the workload.

2. **Memory usage**: Function uses a certain amount of memory for handling the invocation and completing the execution. The maximum amount of memory used during the execution, represents this metric.

3. **Network transmission**: It represents the amount of outgoing network traffic (in bytes) from the platform during the execution of a function.

4. **CPU Usage**: It is the average amount of CPU usage by a function instance when it is executed. It is represented in millicores, where `1000` millicores is equivalent to one vCPU.

9.5 Summary

In this chapter, we explained the methodology used to perform the performance evaluation of the FDN. We introduced nine microbenchmarks, i.e., FaaS functions, and put them under different categories based on their use cases and the system resource they need the most (§9.1.1). We further presented a serverless application with these microbenchmarks (§9.1.2) to test and evaluate the FDN in a real-world scenario. We have employed six heterogeneous clusters in FDN based on different serverless compute platforms spread across the edge-cloud continuum (Table 9.2). We explained about the deployment of the FDN within a Kubernetes cluster (§9.3.1) and created a framework around it for easy evaluation (§9.3.2). The framework replicates the user workload patterns, collects various metrics data, and plots the graphs. In our evaluations, the total test duration for each scenario is fixed to `20 minutes` using the two *Invocations Traces* (R1 and R2, §9.3.2.3). For analyzing the results, we have used different metrics classified into user-centric and platform-centric metrics (§9.4).

10

Function Delivery Network Evaluation Results

“It’s fine to celebrate success but it is more important to heed the lessons of failure."

— Bill Gates

In this chapter, we first present the performance of the individual function microbenchmarks on different clusters in §10.1 and summarize those results in §10.2. After this, we analyze the performance overhead introduced by the FDN’s *Courier Load Balancer* when sending the invocations to the clusters compared to the direct invocation in §10.3. In §10.4, a scenario was devised to confirm the correctness of the set FDN’s function delivery policies, wherein gradual changes are made using *FDN-UI*, and the results of the *Courier Load Balancer* configuration are recorded. In §10.5, we assess the FDN’s bucket replication performance. Following this, we present the performance results of the load balancing algorithms in §10.6. Lastly, in §10.7, we discuss the performance results of the load balancing algorithms.

10.1 FaaS Functions Performance and Resources Usage

In this section, we focus our evaluation on the following three aspects:

- **Functions performance**: FaaS functions, when deployed on heterogeneous clusters spread across the edge-cloud continuum, can behave differently, resulting in performance differences. This difference in behavior can either be due to the different amount of resources available on the clusters or a cluster being optimized/not optimized for certain kind (by kind, we mean CPU-, Memory-, Disk I/O- and Network-intensive function)) of functions. Therefore, we try to answer: *How does the performance of heterogeneous FaaS functions vary when deployed on heterogeneous serverless compute clusters in the continuum?*
- **Resources usage by FaaS functions**: Each FaaS function uses a different amount of resources (CPU, Memory, Disk, and Network Usage) to execute the task. This resource usage can vary with different serverless compute platforms based on their internal implementations and the location of the cluster. Furthermore, each serverless compute platform has different algorithms for scaling the number

of function instances and executing concurrent invocations. This may lead to a varied number of function instances on each cluster, resulting in a higher or lower amount of resource usage. Thus, the question arise, *How does the resource (CPU, Memory, Number of Instances, Disk, and Network Usage) consumption by the functions vary with the change in the clusters and with different serverless compute platforms?*

- **Performance on high user workload**: Certain clusters, such as edge clusters, may have limited resources and, therefore, might not be able to scale well with the increase in user invocations. Furthermore, each serverless compute platform has different algorithms for scaling the number of function instances to handle many concurrent invocations. Therefore, here we try to find the answers to the question: *How does the performance and resources usage vary with the increase in the user workload invocations?*

To this end, we deployed all the function microbenchmarks (§9.1.1) on all the clusters (§9.2). All the FaaS function microbenchmarks except `nodeinfo` and `primes` are allocated with `1024MB` of memory and function concurrency of `50` (the maximum number of concurrent instances allowed for processing events). `nodeinfo` and `primes` functions are allocated with `512MB` of memory and function concurrency of `50`. For each function, the execution duration timeout is set to `50s`. We evaluated the functions on the two *Invocations Traces* (`Trace R1` and `Trace R2`, §9.3.2.3) through the *FDN Test Framework* (§9.3.2) for `20 minutes`.

We now present the performance and resource usage results of each function microbenchmarks on different clusters on two *Invocations Traces* (R1 and R2). For showcasing the performance variation, we analyze *User-Centric* metrics and for resources usage, we analyze *Platform-Centric* metrics (§9.4).

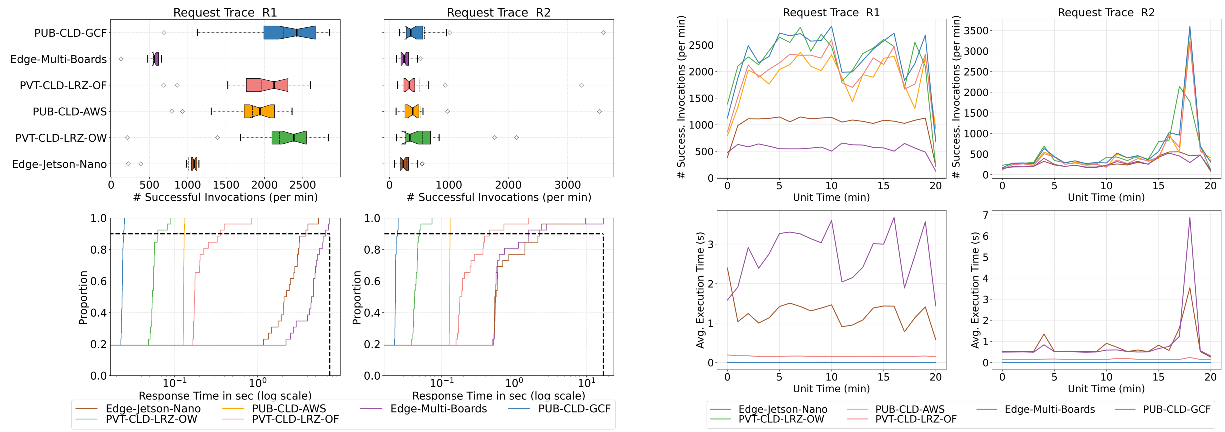
10.1.1 Web-based Function

This category simulates a typical web-based workload that does not require heavy computation. We present here the results of `nodeinfo` function. This function returns the basic characteristics of the node on which it is running, like CPU count, architecture, and uptime.

10.1.1.1 nodeinfo

Figure 10.1 shows the evaluation results of `nodeinfo` function when load tested with two *Invocations Traces* (R1 and R2). Figure 10.1a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' empirical Cumulative Distribution Function (eCDF) plots, showing the distribution of response times of those invocations on the logarithmic scale. eCDF is an estimator of the Cumulative Distribution Function and allows visualizing the distribution of a variable. Figure 10.1b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. We observe the following:

Performance on low and high-workload: For `Trace R2` (low-workload), The *PUB-CLD-GCF* cluster handled the highest number of successful invocations with `590.76` MSI/min with P90 response time of `0.018s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `552.3` MSI/min at P90 response time of `0.03s`. *PUB-CLD-AWS* cluster is not far behind, it handled `529.8` MSI/min with P90 response time of `0.104s`. Among cloud clusters, *PVT-CLD-LRZ-OF* cluster handled the lowest amount of invocations, which handled `500.28` MSI/min at P90 response time of `0.26s`. Two edge clusters handled approximately the same. The *Edge-Jetson-Nano* cluster handled `280.7` MSI/min at P90 response time of `1.08s`, while the *Edge-Multi-Boards* cluster was able to handle `276.3` MSI/min at P90 response time of `1.28s`.



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

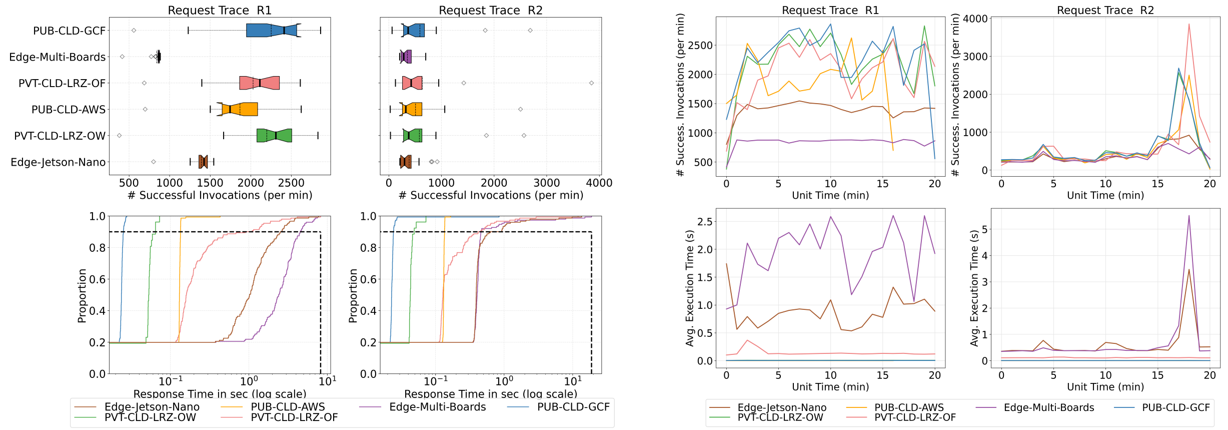
Figure 10.1.: Plots showing the evaluation results of `nodeinfo` when two *Invocations Traces* (R1 and R2) are used for different clusters.

Table 10.1.: Three Platform-Centric metrics showing the mean usage of the resources by `nodeinfo` function on different clusters when handling the two *Invocations Traces*.

Cluster	**Memory Usage (in MB)**		**Instances**		**CPU Usage (millicores)**	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
Edge-Jetson-Nano	32.18	41.4	44.4	11.11	513.3	1271.3
Edge-Multi-Boards	65.24	27.1	42.3	7.56	3939.9	1693.9
PVT-CLD-LRZ-OF	16.19	45.1	46.52	15.09	168.3	476.8
PVT-CLD-LRZ-OW	21.6	22.8	1.45	1.0	71.2	26.1
PUB-CLD-GCF	55.3	52.8	9.6	2.6	-	-
PUB-CLD-AWS	37.9	37.9	38.9	32.6	-	-

For `Trace R1` (high-workload), like for `Trace R2`, the *PUB-CLD-GCF* cluster handled the highest number of successful invocations with `2253.42` MSI/min with P90 response time of `0.019s`. It is closely followed by the two private cloud clusters, the *PVT-CLD-LRZ-OW* cluster handled `2195.53` MSI/min at P90 response time of `0.046s`, while the *PVT-CLD-LRZ-OF* cluster handled `1958.3` MSI/min at P90 response time of `0.195s`. The *PUB-CLD-AWS* cluster handled `1844.28` MSI/min with P90 response time of `0.10s`. The *PUB-CLD-AWS* cluster is deployed in `us-east-1` region while all other clusters are in the Europe region along with the load generation client. Thus, it could be the reason for its higher response time as compared to other cloud clusters. Two edge clusters with low compute resources could not compete well with cloud clusters, but served a higher number of invocations. The *Edge-Jetson-Nano* cluster handled `1012.8` MSI/min with P90 response time of `2.13s`, while the *Edge-Multi-Boards* cluster only handled `550.7` MSI/min with P90 response time of `3.8s`. One can observe that for edge clusters, the P90 response time has risen sharply with the increase in user invocations, while for the other clusters, it has remained approximately the same.

Resources usage: Table 10.1 shows the mean resources' usage (*Platform-Centric* metrics, see §9.4) by `nodeinfo` function on different clusters when handling the two *Invocations Traces* (R1 and R2). From Table 10.1, we can observe that the mean memory usage is almost constant across the two *Invocations*



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.2.: Plots showing the evaluation results of `primes` when two *Invocations Traces* (R1 and R2) are used for different clusters.

Traces for all the clusters except the clusters based on OpenFaaS (*Edge-Jetson-Nano*, *Edge-Multi-Boards*, and *PVT-CLD-LRZ-OF*) where it increased for the *Edge-Multi-Boards* cluster and others it decreased with increase in user invocations. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for all clusters. As the concurrency setting is enabled on the *PVT-CLD-LRZ-OW* cluster, which allows for one function instance to serve many invocations. We can see that it handled the highest number of invocations with just one function instance. CPU usage also followed a similar trend as that of memory usage. The *PVT-CLD-LRZ-OW* cluster, based on the OpenWhisk platform, has the lowest mean memory and CPU usage among all clusters.

10.1.2 CPU-Intensive Functions

This category focuses on CPU-intensive functions, such functions require mainly CPU power and a small amount of memory and network resources [232]. We have considered all three functions under this category:

10.1.2.1 primes

It takes as input an integer `n` and computes the number of primes lower than `n`. Figure 10.2 shows the evaluation results of `primes` function when load tested with two *Invocations Traces* (R1 and R2). Figure 10.2a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.2b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. From Figure 10.2. We observe the following:

Performance on low and high-workload: For `Trace R2` (low-workload), the *PVT-CLD-LRZ-OF* cluster handled the highest number of successful invocations with `635.3` MSI/min at P90 response time of `0.28s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `590.52` MSI/min with P90 response time of

Table 10.2.: Three Platform-Centric metrics showing the mean usage of the resources by `primes-python` function on different clusters when handling the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		CPU Usage (millicores)	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
Edge-Jetson-Nano	33.3	32.7	45.6	11.73	492.7	973.4
Edge-Multi-Boards	52.6	25.2	47.0	8.8	3798.2	1733.5
PVT-CLD-LRZ-OF	18.86	27.06	46.3	5.93	197.7	452.18
PVT-CLD-LRZ-OW	22.5	21.8	1.1	1.0	102.6	28.1
PUB-CLD-GCF	55.5	55.4	9.1	2.7	-	-
PUB-CLD-AWS	36.23	36.2	38.3	33.3	-	-

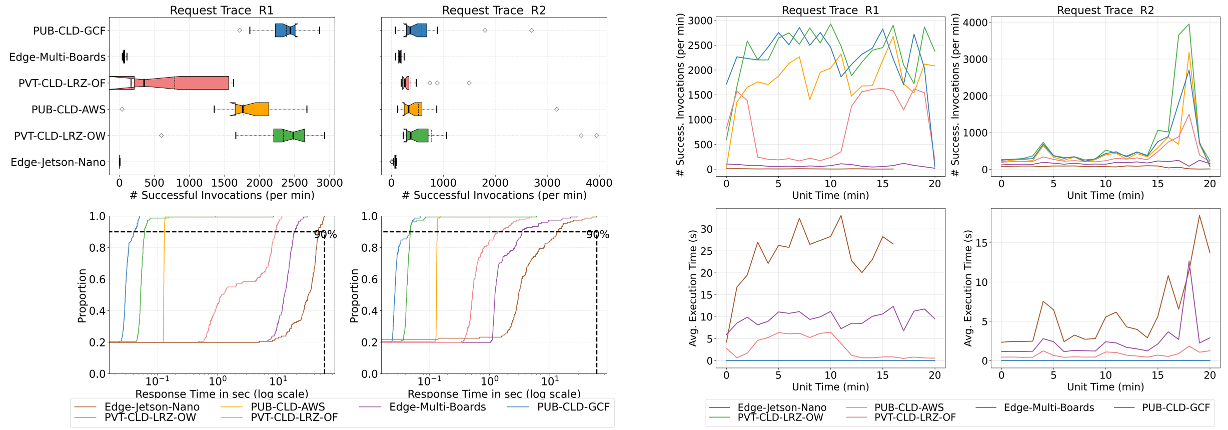
`0.024s`. The *PVT-CLD-LRZ-OW* cluster handled `580.9` MSI/min with P90 response time of `0.036s`. Then comes the *PUB-CLD-AWS* cluster, which handled `508.5` MSI/min with P90 response time of `0.1062s`. Among the edge clusters, the *Edge-Jetson-Nano* cluster handled `385.2` MSI/min with P90 response time of `0.58s` and the *Edge-Multi-Boards* cluster handled `354.2` MSI/min with P90 response time of `0.66s`.

For `Trace R1` (high-workload), the *PUB-CLD-GCF* cluster scaled better and handled the highest number of invocations with `2252.76` MSI/min and P90 response time of `0.019s`. It is closely followed by the two private cloud clusters, the *PVT-CLD-LRZ-OW* cluster handled `2197.6` MSI/min with P90 response time of `0.04s`, while the *PVT-CLD-LRZ-OF* cluster handled `2029.8` MSI/min with P90 response time of `0.38s`. Then comes the *PUB-CLD-AWS* cluster, which handled `1868.0` MSI/min with P90 response time of `0.107s`. Among edge clusters, the edge cluster *Edge-Jetson-Nano* handled `1393.2` MSI/min with P90 response time of `1.19s`, while *Edge-Multi-Boards* cluster handled `840.5` MSI/min with P90 response time of `2.41s`. One can again observe that for edge clusters, the P90 response time has risen with the increase in user invocations, while for the other clusters, it has remained approximately the same.

Resources usage: Table 10.2 shows the mean resources' usage by `primes` function on different clusters when handling the two *Invocations Traces* (R1 and R2). From Table 10.2, we can observe that the mean memory usage is almost constant across the two *Invocations Traces* for all the clusters except the *Edge-Multi-Boards* cluster, where it almost got doubled. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1`. It almost reached the maximum defined value (`50`) for the *Edge-Jetson-Nano*, *Edge-Multi-Boards* and *PVT-CLD-LRZ-OF* clusters. All three clusters are based on OpenFaaS. On the other hand, between the two public cloud clusters, the number of function instances remains almost the same for two traces for the *PUB-CLD-AWS* cluster, while the *PUB-CLD-GCF* cluster in general created a lower number of function instances and was still the best among all clusters. For the *PVT-CLD-LRZ-OW* cluster, the mean memory and CPU usage are the lowest among all clusters. Mean CPU usage for the *Edge-Jetson-Nano* and *PVT-CLD-LRZ-OF* clusters decreased when using the `Trace R1`, since both clusters have a higher number of instances to serve the invocations, resulting in a decrease in mean CPU usage per function instance. While for the *Edge-Multi-Boards* cluster, it increased, even though it has a higher number of function instances. This could be due to the lack of resources available in the cluster, resulting in higher mean CPU usage per instance.

10.1.2.2 linpack

It solves a dense linear system of equations in double precision and returns the results in GFlops. Figure 10.3 shows the results of `linpack` function when load tested with two *Invocations Traces* (R1 and R2).



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.3.: Plots showing the evaluation results of `linpack` when two *Invocations Traces* (R1 and R2) are used for different clusters.

Figure 10.3a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.3b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. From Figure 10.3. We observe the following:

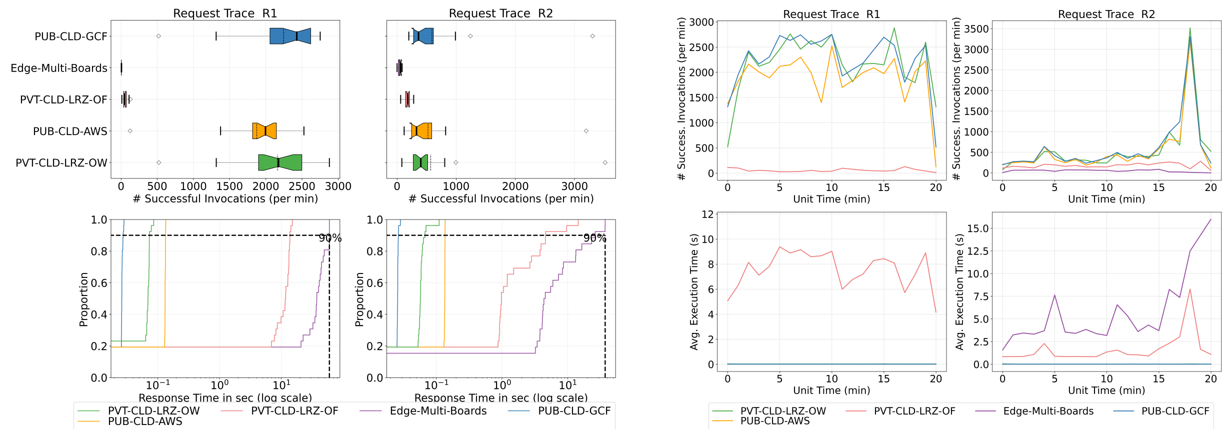
Performance on low and high-workload: For `Trace R2` (low-workload), the *PVT-CLD-LRZ-OW* cluster handled the highest number of successful invocations with `574.07` MSI/min at P90 response time of `0.075s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `587.1` MSI/min with P90 response time of `0.025s`. Then comes the *PUB-CLD-AWS* cluster, which handled `522.90` MSI/min) at a P90 response time of `0.10s`. Private cloud cluster *PVT-CLD-LRZ-OF* handled `327.8` MSI/min with a P90 response time of `0.70s`. Among the edge clusters, the *Edge-Jetson-Nano* cluster handled `70.4` MSI/min with P90 response time of `5.5s` and the *Edge-Multi-Boards* cluster handled `167.6` MSI/min with P90 response time of `2.09s`.

For `Trace R1`, the *PUB-CLD-GCF* cluster handled the highest number of invocations with `2381.3` MSI/min and P90 response time of `0.026s`. It is closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled `2337.5` MSI/min at P90 response time of `0.100s`. Then comes the *PUB-CLD-AWS* cluster, which handled `1776.5` MSI/min at P90 response time of `0.109s`. Out of all the cloud clusters, the *PVT-CLD-LRZ-OF* cluster performed the worst, which handled `788.2` MSI/min at P90 response time of `3.6s`. Even though the *PVT-CLD-LRZ-OF* cluster has the same amount of resources as the *PVT-CLD-LRZ-OW* cluster, it could not perform comparably to it. This can be attributed to the different serverless compute platforms used in these clusters. Additionally, the function image differs for two clusters, which could also be the reason for the slow performance of the *PVT-CLD-LRZ-OF* cluster. Two edge clusters with low compute resources could not compete well with cloud clusters and perform poorly compared to them. The *Edge-Multi-Boards* cluster handled `69.5` MSI/min at P90 response time of `11.6s`, while the *Edge-Jetson-Nano* cluster handled `4.45` MSI/min at P90 response time of `27.9s`.

Resources usage: Table 10.3 shows the mean resources' usage by `linpack` function on different clusters when handling the two *Invocations Traces* (R1 and R2). From Table 10.3, we can observe that the mean memory usage is almost constant across the two traces for all the clusters except the clusters based on

Table 10.3.: Platform-Centric metrics showing the mean usage of the resources by `linpack` function on different clusters when handling the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		CPU Usage (millicores)	
	R1	R2	R1	R2	R1	R2
Edge-Jetson-Nano	549.3	149.2	1.0	1.0	3676.4	3485.1
Edge-Multi-Boards	266.0	56.17	1.0	1.0	8413.1	4300.1
PVT-CLD-LRZ-OF	291.7	55.7	19.6	9.2	2241.9	2153.8
PVT-CLD-LRZ-OW	29.8	29.19	1	1	133.42	30.9
PUB-CLD-GCF	76.0	78.40	6.9	2.5	-	-
PUB-CLD-AWS	70.5	70.5	39.1	33.02	-	-



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.4.: Plots showing the evaluation results of `sentiment-analysis` when two *Invocations Traces* (R1 and R2) are used for different clusters.

OpenFaaS, where it increased substantially with an increase in user invocations. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for the *PUB-CLD-AWS*, *PUB-CLD-GCF*, and *PVT-CLD-LRZ-OF* clusters, while for others it remains at `1` only. The *PVT-CLD-LRZ-OW* cluster can serve the highest number of invocations with just one function instance. For the *PVT-CLD-LRZ-OW* cluster, the mean memory and CPU usage are the lowest among all clusters. Mean CPU usage for all clusters increased with an increase in user invocations.

10.1.2.3 sentiment-analysis

It analyzes the sentiment of a provided string using the Python TextBlob library. Figure 10.4 shows the results of `sentiment-analysis` function when load tested with two *Invocations Traces* (R1 and R2). Figure 10.4a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.4b shows

Table 10.4.: Platform-Centric metrics showing the mean usage of the resources by `sentiment-analysis` function on different clusters when handling the two *Invocations Traces*.

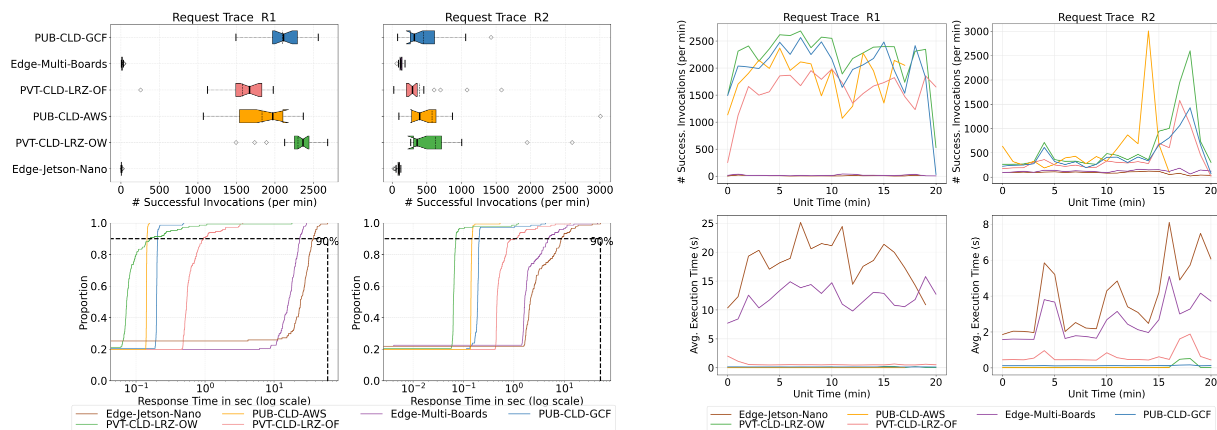
Cluster	Memory Usage (in MB)		Instances		CPU Usage (millicores)	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
Edge-Multi-Boards	201.48	136.9	1.0	1.0	3751.24	4760.75
PVT-CLD-LRZ-OF	685.18	234.9	1.0	1.0	3668.6	2384.6
PVT-CLD-LRZ-OW	41.98	41.94	1.0	1.0	347.4	107.5
PUB-CLD-GCF	140.8	139.9	9	2.6	-	-
PUB-CLD-AWS	117.3	117.2	41.8	41.9	-	-

the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. This function was not deployed on the *Edge-Jetson-Nano* cluster. Since this function requires downloading Natural Language Toolkit (NLTK) data from the internet, that cluster does not have access to the internet. From Figure 10.4. We observe the following:

Performance on low and high-workload: For `Trace R2` (low-workload), the *PUB-CLD-GCF* cluster handled the highest number of successful invocations, which handled `589.0` MSI/min, closely followed by the *PVT-CLD-LRZ-OW* cluster with `567.53` MSI/min and the *PUB-CLD-AWS* cluster with `523.6` MSI/min. This trend continues in their P90 response time, the *PUB-CLD-GCF* cluster (`0.021s`) has the lowest P90 response time, then comes the *PVT-CLD-LRZ-OW* cluster with P90 response time of `0.049s` and then the *PUB-CLD-AWS* cluster (`0.109s`). The *PVT-CLD-LRZ-OF* cluster handled `178.4` MSI/min with P90 response time of `2.30s`. The edge cluster *Edge-Multi-Boards* handled `49.9` MSI/min with P90 response time of `9.3s`.

For `Trace R1` (high-workload), the *PUB-CLD-GCF* cluster handled the highest number of invocation, which handled `2245.8` MSI/min with the lowest P90 response time of `0.021s`. It is again closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled `2160.4` MSI/min with P90 response time of `0.055s`. The *PUB-CLD-AWS* cluster comes next with P90 response time of `0.107s` and handled `1875.0` MSI/min. Private cloud cluster *PVT-CLD-LRZ-OF* could only handle `56.4` MSI/min with P90 response time of `9.5s`. It clearly shows the overhead introduced by the OpenFaaS platform compared to the OpenWhisk serverless compute platform, since both clusters have the same amount of resources. The edge cluster *Edge-Multi-Boards* with low compute resources could not perform well for this function (`4.3` MSI/min with P90 response time `34.5s`).

Resources usage: Table 10.4 shows the mean resources' usage by `sentiment-analysis` function on different clusters when handling the two *Invocations Traces*. From Table 10.4, we can observe that the mean memory usage is almost constant across the two traces for all the clusters except the clusters based on OpenFaaS (*Edge-Multi-Boards*, and *PVT-CLD-LRZ-OF*) where it increased substantially with an increase in user invocations. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters, while for others, it remains at `1` only. The *PVT-CLD-LRZ-OW* cluster can serve a high number of invocations with just one function instance. The mean CPU usage per function instance for clusters increased with `Trace R1` except for the *Edge-Multi-Boards* cluster, where it decreased as most of the invocations failed.



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.5.: Plots showing the evaluation results of `dd` when two *Invocations Traces* (R1 and R2) are used for different clusters.

10.1.3 Memory- and Disk-Intensive Functions

10.1.3.1 dd

This function utilizes the Unix command-line utility dd - that stands for "Data Definition" [177]. Its main usage is copying files from different locations, with the advantage that it is more efficient than other commands when dealing with large documents such as device files. Figure 10.5 shows the results of `dd` function when load tested with two *Invocations Traces* (R1 and R2). Figure 10.5a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.5b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. From Figure 10.5. We observe the following:

Performance on low and high-workload: For `Trace R2`, the *PVT-CLD-LRZ-OW* cluster handled the highest number of successful invocations with `622.5` MSI/min at P90 response time of `0.073s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `574.8` MSI/min at P90 response time of `0.11s`. Then comes the *PUB-CLD-GCF* cluster, which handled `455.14` MSI/min at P90 response time of `0.22s`. *PVT-CLD-LRZ-OF* cluster handled `397.4` MSI/min at P90 response time of `0.68s`. Among the edge clusters, the *Edge-Multi-Boards* cluster handled `126.36` MSI/min at P90 response time of `2.5s`, while the *Edge-Jetson-Nano* cluster handled `89.76` MSI/min at P90 response time of `3.9s`.

For `Trace R1` (high-workload), the *PVT-CLD-LRZ-OW* cluster handled the highest number of invocations, which handled `2300.8` MSI/min at P90 response time of `0.21s`. It is closely followed by the *PUB-CLD-GCF* cluster, which handled `2099.6` MSI/min with the P90 response time of `0.16s`. The *PUB-CLD-AWS* cluster has the lowest P90 response time of `0.11s` (which can also been seen in Figure 10.5a) and handled `1833.6` MSI/min. Private cloud cluster *PVT-CLD-LRZ-OF* could handle `1580.0` MSI/min at P90 response time of `0.6s`. It clearly shows the overhead introduced by the OpenFaaS platform compared to the OpenWhisk serverless compute platform. Two edge clusters with low compute resources could not perform well for this

Table 10.5.: Platform-Centric metrics showing the mean usage of the resources by `dd` function on different clusters when handling the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		Write IOPS	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
Edge-Jetson-Nano	401.5	139.7	1.0	1.0	2.39	3.17
Edge-Multi-Boards	250.9	73.46	1.0	1.0	9.46	8.0
PVT-CLD-LRZ-OF	26.1	215.9	46.5	8.9	1.4	5.96
PVT-CLD-LRZ-OW	31.0	29.1	1.14	1.21	66.8	12.15
PUB-CLD-GCF	83.0	82.92	11.5	6.24	-	-
PUB-CLD-AWS	48.5	47.9	8.5	17.7	-	-

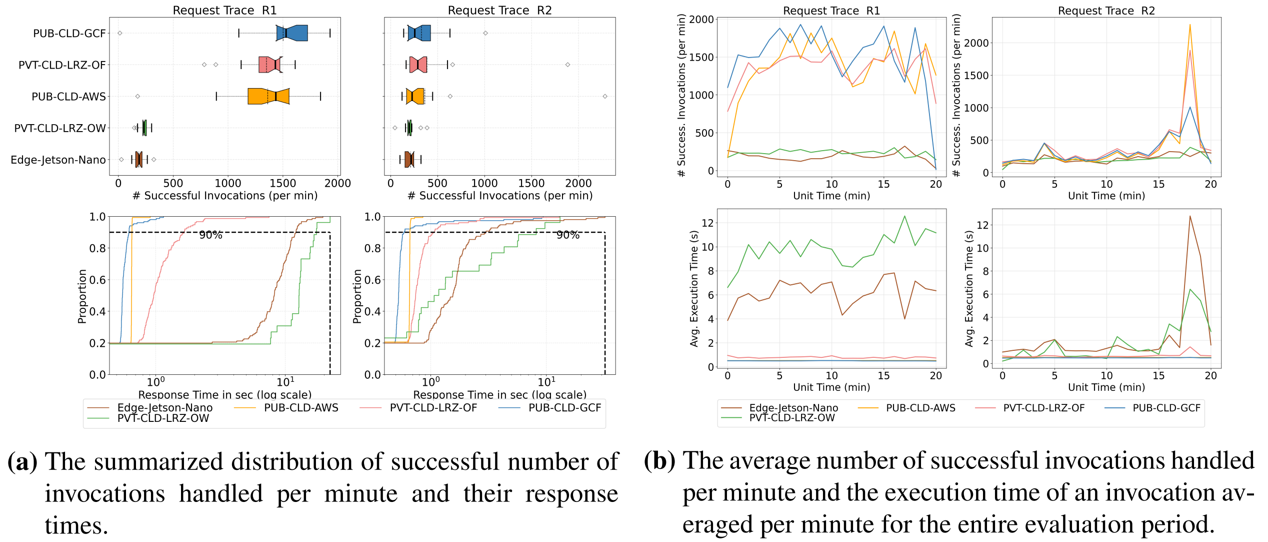
function. The *Edge-Multi-Boards* cluster handled `17.14` MSI/min at P90 response time of `14.5s`, while the *Edge-Jetson-Nano* cluster handled `6.8` MSI/min at P90 response time of `21.2s`.

Resources usage: Table 10.5 shows the mean resources' usage by `dd` function on different clusters when handling the two *Invocations Traces* (R1 and R2). From Table 10.5, we can observe that the mean memory usage is almost constant across the two traces for all the clusters, except for the clusters based on OpenFaaS. For edge clusters, it increased substantially, while for the *PVT-CLD-LRZ-OF* cluster, the mean memory usage decreased with an increase in user invocations. This can be attributed to the high number of function instances created in *PVT-CLD-LRZ-OF* cluster; as a result, the mean memory usage per function instance is lower. The *PVT-CLD-LRZ-OW* cluster has the least memory consumption among all clusters. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for the *PVT-CLD-LRZ-OF* and *PUB-CLD-GCF* clusters. For both edge clusters, the mean number of function instances remained at one, which could be attributed to fewer resources and hence the inability to scale. Additionally, the number of I/O operations performed by the *PVT-CLD-LRZ-OW* cluster is much higher than the other clusters, as it has served a higher number of invocations than the other clusters.

10.1.3.2 gzip-compression

This function takes as input an integer *file_size*, writes *file_size* GB on a file on the disk and then read the same amount in the memory. Figure 10.6 shows the results of `gzip-compression` function when load tested with two *Invocations Traces* (R1 and R2). Figure 10.6a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.6b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. This function was not deployed on the *Edge-Multi-Boards* cluster because this function requires a high amount of resources and would overload the cluster. From Figure 10.6. We observe the following:

Performance on low and high-workload: For `Trace R2`, *PVT-CLD-LRZ-OF* cluster handled the highest number of successful invocations, which handled `386.9` MSI/min at P90 response time of `0.78s`. It is closely followed by the *PUB-CLD-AWS* cluster, which handled `363.5` MSI/min at P90 response time of `0.52s`. Then comes the *PUB-CLD-GCF* cluster, which handled `328.5` MSI/min at P90 response time of `0.63s`. The *PVT-CLD-LRZ-OW* cluster handle `203.61` MSI/min at P90 response time of `2.5s`. The edge



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.6.: Plots showing the evaluation results of `gzip-compression` when two *Invocations Traces* (R1 and R2) are used for different clusters.

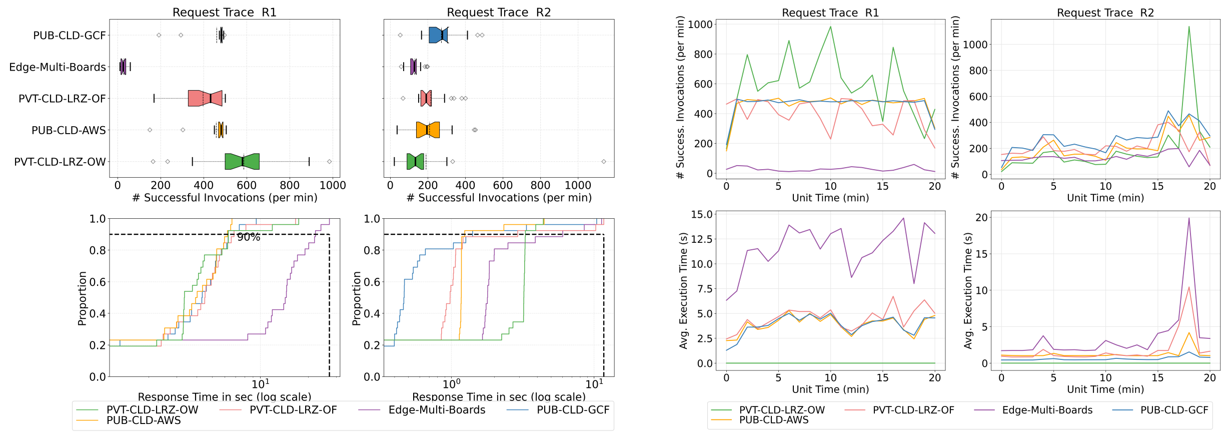
Table 10.6.: Platform-Centric Metrics showing the mean usage of the resources by `gzip-compression` function on different clusters when handling the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		Write IOPS	
	R1	R2	R1	R2	R1	R2
Edge-Jetson-Nano	307.47	69.5	4.11	1.0	21.4	22.0
PVT-CLD-LRZ-OF	31.7	47.6	46.52	8.9	11.5	33.1
PVT-CLD-LRZ-OW	34.37	34.6	1.83	1.35	52.6	80.5
PUB-CLD-GCF	116.0	115.2	28.0	8.3	-	-
PUB-CLD-AWS	51.4	51.3	54.67	83.12	-	-

cluster *Edge-Jetson-Nano* handled more number of invocations than that of the *PVT-CLD-LRZ-OW* cluster, i.e., `211.71` MSI/min at a lower P90 response time of `2.04s`.

For `Trace R1` (high-workload), the *PUB-CLD-GCF* cluster handled the highest number of invocation, which handled `1502.1` MSI/min at P90 response time of `0.474s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `1361.9` MSI/min at P90 response time of `0.525s`. Private cloud cluster *PVT-CLD-LRZ-OF* could handle `1350.8` MSI/min at P90 response time of `0.99s`. However, the *PVT-CLD-LRZ-OW* cluster could not perform well for this function and hence handled only `233.60` MSI/min with P90 response time of `10.94s`. The edge cluster *Edge-Jetson-Nano* has again a lower P90 response time of `7.36s` than that of the *PVT-CLD-LRZ-OW* cluster, and it handled `186.2` MSI/min.

Resources usage: Table 10.6 shows the mean resources' usage by `gzip-compression` function on different clusters when handling the two *Invocations Traces*. From Table 10.6, we can observe that the mean memory usage is almost constant across the two traces for all the clusters except the clusters based on OpenFaaS. For the *Edge-Jetson-Nano* cluster, it increased substantially with an increase in user invocations, while for the *PVT-CLD-LRZ-OF* cluster, it decreased since more function instances are created. As a result, the load on a function instance is less, resulting in decreased memory usage. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for all the clusters. The



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.7.: Plots showing the evaluation results of `json-loads` when two *Invocations Traces* (R1 and R2) are used for different clusters.

limit for maximum function instances was set to 50 and still, the *PUB-CLD-AWS* cluster scaled beyond it. Additionally, the number of write IOPS done by each function instance for each cluster is decreased with the increase in user invocations. This could be attributed to the fact that, with more instances, each instance will handle fewer invocations, resulting in a lower number of write IOPS by each function instance.

10.1.4 Network-Intensive Function

10.1.4.1 json-loads

As the name suggests, this function sends a request to a remote URL containing a JSON file and waits for the response. Then, it loads the file, converts it into a string, and returns the time needed to perform all the operations. Figure 10.7 shows the results of `json-loads` function when load tested with two *Invocations Traces*. Figure 10.6a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.7b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. Since this function requires accessing a JSON file from the internet and the *Edge-Jetson-Nano* cluster does not have internet access; therefore it was not deployed on it. From Figure 10.7. We observe the following:

Performance on low and high-workload: For `Trace R2`, the *PUB-CLD-GCF* cluster handled the highest number of successful invocations, with `272.1` MSI/min at P90 response time of `0.97s`. It is followed by the *PVT-CLD-LRZ-OF* cluster, which handled `215.48` MSI/min at P90 response time of `1.61s`. Then comes the *PUB-CLD-AWS* cluster, which handled `209.0` MSI/min at P90 response time of `1.07s`. They were followed by the *PVT-CLD-LRZ-OW* cluster, which handled `190.3` MSI/min at P90 response time of `2.48s`. The edge cluster *Edge-Multi-Boards* could handle `129.0` MSI/min at P90 response time of `2.27s`.

For `Trace R1` (high-workload), interestingly the slowest cluster in `Trace 2`, i.e, the *PVT-CLD-LRZ-OW* cluster handled the highest number of invocations with `586.5` MSI/min at P90 response time of `4.13s`.

Table 10.7.: Platform-Centric metrics showing the mean usage of the resources by `json-loads` function on different clusters serving the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		NW Transmit Rate (KB/s)	
	R1	**R2**	**R1**	**R2**	**R1**	**R2**
Edge-Multi-Boards	167.9	77.7	1.0	1.0	50.91	43.1
PVT-CLD-LRZ-OF	160.0	88.11	24.5	3.09	56.4	43.4
PVT-CLD-LRZ-OW	10.6	6.97	1.0	1.0	49.16	41.3
PUB-CLD-GCF	56.7	57.4	43.9	9.714	348.7	205.8
PUB-CLD-AWS	40.4	40.14	40.88	30.18	-	-

It is followed by two public cloud clusters, the *PUB-CLD-GCF* cluster handled `460.6` MSI/min at P90 response time of `3.89s`, while the *PUB-CLD-AWS* cluster handled `458.3` MSI/min at P90 response time of `3.56s`. Both the public cloud clusters have a lower P90 response time than the *PVT-CLD-LRZ-OW* cluster. Private cloud cluster *PVT-CLD-LRZ-OF* could handle `397.50` MSI/min with P90 response time of `4.29s`. The OpenWhisk-based cluster performed better here than the OpenFaaS-based clusters. The edge cluster *Edge-Multi-Boards*, could only handle `27.9` MSI/min with P90 response time of `12.5s`.

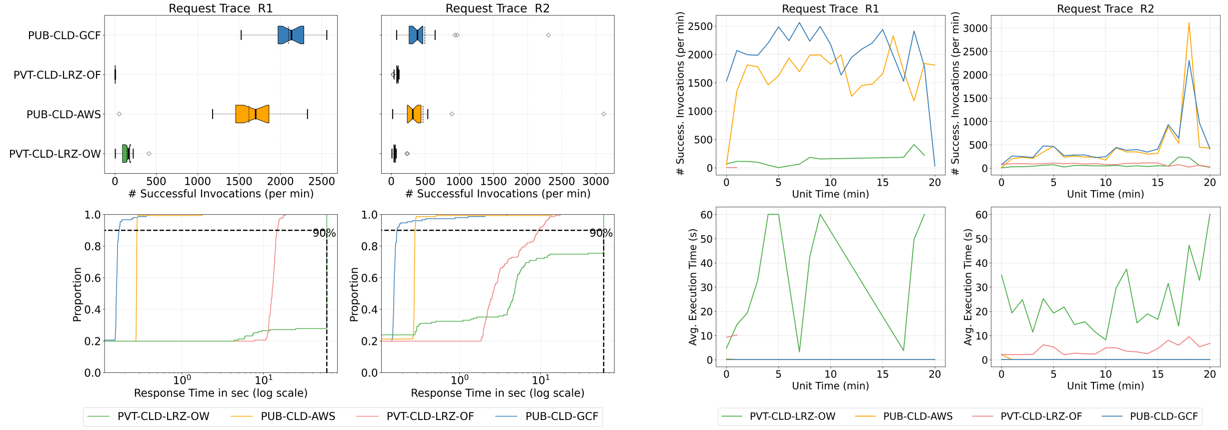
Resources usage: Table 10.7 shows the mean resources' usage by `json-loads` function on different clusters when handling the two *Invocations Traces*. From Table 10.7, we can observe that the mean memory usage is almost constant across the two traces for the two public cloud clusters, while for others, it increased with an increase in user workload. It almost got doubled for the clusters based on OpenFaaS. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for all clusters except the *PVT-CLD-LRZ-OW* and *Edge-Multi-Boards*. *PVT-CLD-LRZ-OW* is enabled with a function instance to handle concurrent invocations. Thus, this function does not require scaling. While the *Edge-Multi-Boards* cluster has low computing resources, it did not scale. Additionally, since this function involves downloading the file from the internet, we can see network transmission rise when load tested with `Trace R1` for all clusters.

10.1.5 ML-Based Functions

10.1.5.1 lr-prediction

This function first downloads a linear regression model trained on user reviews data and the test data from the storage buckets located on the GCP. It then performs prediction using the downloaded model on the test data. Figure 10.8 shows the results of `lr-prediction` function when load tested with two *Invocations Traces*. Figure 10.8a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.8b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. This function was only deployed on cloud clusters due to its high resource requirements. From Figure 10.8. We observe the following:

Performance on low and high-workload: For `Trace R2`, the *PUB-CLD-GCF* cluster handled the highest number of successful invocations with `490.4` MSI/min at P90 response time of `0.18s`. It is followed by the



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.8.: Plots showing the evaluation results of `lr-prediction` when two *Invocations Traces* (R1 and R2) are used for different clusters.

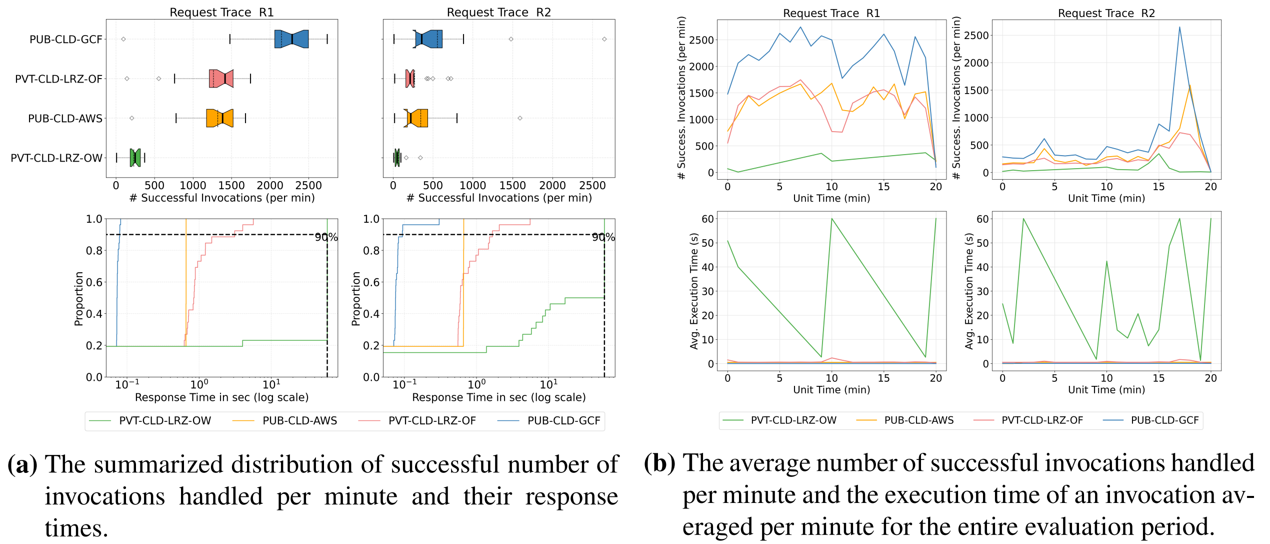
Table 10.8.: Platform-Centric metrics showing the mean usage of the resources by `lr-Prediction` function on different clusters serving the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		NW Transmit Rate (KB/s)	
	R1	R2	R1	R2	R1	R2
PVT-CLD-LRZ-OF	683.6	301.5	1.0	1.0	4.5	13.9
PVT-CLD-LRZ-OW	42.8	65.6	5.78	6.72	2.45	1.29
PUB-CLD-GCF	201.8	213.8	8.9	6.8	0.009	0.009
PUB-CLD-AWS	160.9	130.9	5.33	2.73	-	-

PUB-CLD-AWS cluster with `466.09` MSI/min at P90 response time of `0.31s`. Both private cloud clusters could not handle the workload for this function. The *PVT-CLD-LRZ-OF* cluster handled only handle `84.3` MSI/min at P90 response time of `3.79s`, while the *PVT-CLD-LRZ-OW* cluster handled only `61.8` MSI/min at P90 response time of `17.2s`.

For `Trace R1` (high-workload), the *PUB-CLD-GCF* cluster handled the highest number of invocations with `2099.0` MSI/min at P90 response time of `0.13s`. It is closely followed by the *PUB-CLD-AWS* cluster, which handled `1620.5` MSI/min at P90 response time of `0.23s`. The *PUB-CLD-GCF* cluster has the lowest P90 response time. Since this function requires downloading the model from the storage bucket along with the data, and that process takes more than the set time limit for most invocations. As a result, both private cloud clusters could not handle the high workload for this function and hence were only able to serve a low number of successful invocations. Furthermore, this is not observed in the public cloud platforms, as they do model and data caching and hence could serve the invocations easily. The *PVT-CLD-LRZ-OF* cluster could handle `3.2` MSI/min at P90 response time of `44.0s`, while the *PVT-CLD-LRZ-OW* cluster could handle `144.8` MSI/min at P90 response time of `10.6s`.

Resources usage: Table 10.8 shows the mean resources' usage by `lr-prediction` function on different clusters when handling the two *Invocations Traces* (R1 and R2). From Table 10.8, we can observe that the mean memory usage varies across the clusters, with the *PVT-CLD-LRZ-OW* cluster's function instances



(a) The summarized distribution of successful number of invocations handled per minute and their response times.

(b) The average number of successful invocations handled per minute and the execution time of an invocation averaged per minute for the entire evaluation period.

Figure 10.9.: Plots showing the evaluation results of `image-processing` when two *Invocations Traces* (R1 and R2) are used for different clusters.

using the least memory. Furthermore, the mean number of function instances created to handle the user invocations increased with `Trace R1` for all public cloud clusters. For the *PVT-CLD-LRZ-OF* cluster, the mean number of function instances remained at one only, while for the *PVT-CLD-LRZ-OW* cluster, the mean number of function instances remains almost the same across two traces. Network usage by the *PUB-CLD-GCF* cluster is the lowest compared to other clusters, since the buckets to fetch the data and model reside within the Google cloud.

10.1.5.2 image-processing

It inputs the image name and object storage (here MinIO) credentials. Based on it, it downloads the image from the object storage and performs basic image operations flip, rotate, filter, grayscale, and resize the image. Figure 10.9 shows the results of `image-processing` function when load tested with two *Invocations Traces*. Figure 10.9a presents the two box plots, showing the distribution of the successful number of invocations handled per minute by each cluster. The figure also shows the corresponding response times' eCDF plots, showing the distribution of response times of those invocations on the logarithmic scale. Figure 10.9b shows the corresponding plots for the average number of successful invocations handled per minute during the entire evaluation period, along with the execution time of an invocation averaged per minute. This function was also only deployed on cloud clusters due to its high resource requirements. From Figure 10.9. We observe the following:

Performance on low and high-workload: For `Trace R2`, the *PUB-CLD-GCF* cluster handled the highest number of successful invocations with `554.9` MSI/min at P90 response time of `0.07s`. It is followed by the *PUB-CLD-AWS* cluster with `346.19` MSI/min at P90 response time of `0.07s`. Among private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled `268.7` MSI/min at P90 response time of `0.88s`, while the *PVT-CLD-LRZ-OW* cluster could not perform well on this function and handled only `65.2` MSI/min at P90 response time of `32.59s`.

For `Trace R1` (high-workload), the *PUB-CLD-GCF* cluster handled the highest number of invocations with `2148.2`) MSI/min at P90 response time of `0.06s`. It is closely followed by the *PUB-CLD-AWS* cluster with

Table 10.9.: Platform-Centric Metrics showing the mean usage of the resources by `image-processing` function on different clusters serving the two *Invocations Traces*.

Cluster	Memory Usage (in MB)		Instances		NW Transmit Rate (KB/s)	
	R1	R2	R1	R2	R1	R2
PVT-CLD-LRZ-OF	56.6	79.1	34.23	13.30	6.4	11.65
PVT-CLD-LRZ-OW	14.03	17.7	5.50	6.2	0.99	0.48
PUB-CLD-GCF	65.5	70.41	8.9	5.20	1528.9	596.2
PUB-CLD-AWS	49.5	49.4	7.14	7.17	-	-

`1321.00` MSI/min at P90 response time of `0.53s`. Since this function requires downloading the image from the MinIO bucket, which is deployed on an instance on Google Cloud, hence the *PUB-CLD-GCF* cluster has the lowest P90 response time. Among the private cloud clusters, the *PVT-CLD-LRZ-OW* cluster could not handle high workload for this function and hence was able to only handle `231.5` MSI/min with substantial high P90 response time of `46.3s`. However, the *PVT-CLD-LRZ-OF* cluster was able to handle `1265.20` MSI/min with P90 response time of `1.10s`). It again shows that, OpenWhisk-based platform is not well suited for this type of functions.

Resources usage: Table 10.9 shows the mean resources' usage by `image-processing` function on different clusters when handling the two *Invocations Traces*. From Table 10.8, we can observe that the mean memory usage remains almost the same across two traces for all the clusters. The *PVT-CLD-LRZ-OW* cluster's function instances use the least memory, since most of the invocations resulted in an error. Furthermore, the mean number of function instances created to handle the user invocations from `Trace R1` either increased or remained the same as with `Trace R2` for all clusters. Network usage by the *PUB-CLD-GCF* cluster is highest compared to other clusters since the image needs to be fetched for every invocation. Additionally, it served the most invocations with a low number of instances. Hence per instance, network usage is high.

10.2 FaaS Functions Performance and Resources Usage Summary

Serverless compute clusters part of FDN can be made up of different serverless compute platforms and deployed at different locations. Most open-source platforms are based on Kubernetes, where they start a function in a container, while the public cloud provider's platform starts a function in a MicroVM. Thus the performances of the functions within the clusters based on these platforms will vary. Furthermore, each cluster may not have the same resources as others. For example, edge clusters may have limited resources compared to cloud clusters. In Table 10.10, we present the summarized results of the average number of invocations made per minute by each cluster for each FaaS function when load tested with two `Invocation Traces`. We also highlight the cluster for each function and invocation traces, which has made the highest number of successful invocations. One can observe that across all the functions, *PUB-CLD-GCF* handled the most number of invocations. However for some functions, the other clusters performed the best. In this section, we summarize the performance and resource usage of the FaaS functions on various clusters.

nodeinfo: For this function, we can say that all clusters scaled better with the increase in user invocations. Among cloud clusters, the *PUB-CLD-GCF*, *PUB-CLD-AWS*, and *PVT-CLD-LRZ-OW* clusters were able to handle the user invocations with the consistent P90 response times. The *PUB-CLD-GCF* cluster handled the

Table 10.10.: Summary of the average number of invocations made per minute by each cluster for each FaaS function when load tested with two `Invocation Traces`.

Function	-	PUB-CLD-GCF	PUB-CLD-AWS	PVT-CLD-LRZ-OW	PVT-CLD-LRZ-OF	Edge-Jetson-Nano	Edge-Multi-Boards
`nodeinfo`	R2	**2253.42**	1844.28	2195.53	1958.3	1012.8	550.7
	R1	**590.76**	529.8	552.3	500.28	280.7	276.3
`primes`	R1	**2252.76**	1868.0	2197.6	2029.8	1393.2	840.5
	R2	**590.52**	508.5	580.9	635.3	385.2	354.2
`linpack`	R1	**2381.3**	1776.5	2337.5	788.2	4.45	69.5
	R2	**587.1**	522.90	574.07	327.8	70.4	167.6
`sentiment`	R1	**2245.8**	1875.0	2160.4	56.4	-	4.3
	R2	**589.0**	523.6	567.53	178.4	-	49.9
`dd`	R1	2099.6	1833.6	**2300.8**	1580.0	6.8	17.14
	R2	455.14	574.8	**622.5**	397.4	89.76	126.36
`gzip`	R1	**1502.1**	1361.9	233.60	1350.8	186.2	-
	R2	328.5	363.5	203.61	**386.9**	211.71	-
`json-loads`	R1	460.6	458.3	**586.5**	397.50	-	27.9
	R2	**272.1**	209.0	190.3	215.48	-	129.0
`lr-prediction`	R1	**2099.0**	1620.5	144.8	3.2	-	-
	R2	**490.4**	466.09	61.8	84.3	-	-
`image-process`	R1	**2148.2**	1321.00	231.5	1265.20	-	-
	R2	**554.9**	346.19	65.2	268.7	-	-

most number of invocations. Edge clusters also handled the invocations well for this function, but they are better suited if the number of invocations is low. Furthermore, the mean memory usage is almost constant across the two *Invocations Traces* for all the clusters except the clusters based on OpenFaaS. The mean number of function instances created to handle the user invocations increased with `Trace R1` for all clusters except for the *PVT-CLD-LRZ-OW* cluster, where it stayed as one. As the concurrency setting is enabled on the *PVT-CLD-LRZ-OW* cluster, which allows for one function instance to serve many invocations. We can see that it handled the highest number of invocations with just one function instance. CPU usage also followed a similar trend as that of memory usage.

primes: For this function, we can say that the *PUB-CLD-GCF* cluster performed better with the increase in user invocations as compared to other clusters. Edge clusters performed well with the increase in number of invocations, but they are better suited if the number of invocations are low. Both public cloud clusters, and the *PVT-CLD-LRZ-OW* cluster, were able to serve the user invocations with the consistent P90 response times. Among private cloud clusters, *PVT-CLD-LRZ-OW* performed better than the *PVT-CLD-LRZ-OF*. Memory usage is almost constant even with the increase in user invocations for all the clusters except *Edge-Multi-Boards*, where it almost got doubled. The mean number of function instances created to handle the user invocations increased with `Trace R1`. It almost reached the maximum defined value (`50`) for all the clusters based on OpenFaaS.

linpack: For this function, *PVT-CLD-LRZ-OW* and *PUB-CLD-GCF* performed better compared to other clusters. The three cloud clusters: *PUB-CLD-GCF*, *PUB-CLD-AWS* and *PVT-CLD-LRZ-OW* were able to serve the invocations with consistent P90 response times. Edge clusters are better suited if the number of invocations is low; they could not scale well for this function when the number of invocations is increased.

OpenFaaS clusters performed the worst compared to other serverless compute platforms due to their inability to scale well with the user invocations. Furthermore, the mean memory usage is almost constant across the two traces for all the clusters except the clusters based on OpenFaaS. The mean number of function instances created to handle the invocations increased with `Trace R1` for most of the clusters, except the edge clusters and *PVT-CLD-LRZ-OW*. For the *PVT-CLD-LRZ-OW* cluster, the mean memory and CPU usage are the lowest among all clusters.

sentiment-analysis: This function was not deployed on the *Edge-Jetson-Nano* cluster. Since this function requires downloading NLTK data from the internet, that cluster does not have access to the internet. For this function, *PVT-CLD-LRZ-OW* and *PUB-CLD-GCF* performed better compared to other clusters. The *Edge-Multi-Boards* cluster and *PVT-CLD-LRZ-OF* are not suited for this function. These clusters are based on the OpenFaaS; therefore, it could be due to the serverless compute platform that they could not perform well. The three cloud clusters: *PUB-CLD-GCF*, *PUB-CLD-AWS* and *PVT-CLD-LRZ-OW* were able to serve the invocations with consistent P90 response times. Furthermore, the mean memory usage is almost constant across the two traces for all the clusters except the clusters based on OpenFaaS. The mean number of function instances created to handle the invocations increased with `Trace R1` for both public cloud clusters, while it remained at one for others. The mean CPU usage per function instance for clusters increased with `Trace R1` except for the *Edge-Multi-Boards* cluster, where it decreased as most of the invocations failed.

dd: Similar to the *linpack* function, in this function also, the *PVT-CLD-LRZ-OW* cluster, *PUB-CLD-GCF* and *PUB-CLD-AWS* scaled better with the increase in user invocations as compared to other clusters. All these clusters could serve the user invocations with consistent P90 response times. Additionally, *PVT-CLD-LRZ-OF* is not far behind these clusters, while edge clusters are better suited if the number of invocations is low. The mean memory usage is almost constant across the two traces for all the clusters, except those based on OpenFaaS. For edge clusters, it increased substantially, while for the *PVT-CLD-LRZ-OF* cluster, the mean memory usage decreased with an increase in user invocations. This can be attributed to the high number of function instances created in *PVT-CLD-LRZ-OF* cluster; as a result, the mean memory usage per function instance is lower. The *PVT-CLD-LRZ-OW* cluster has the least memory consumption among all clusters. Additionally, the number of I/O operations performed by the *PVT-CLD-LRZ-OW* cluster is much higher than the other clusters, as it has served a higher number of invocations than the other clusters.

gzip-compression: This function was not deployed on the *Edge-Multi-Boards* cluster because this function requires a high amount of resources and would overload the cluster. For this function, the *PVT-CLD-LRZ-OF* cluster, *PUB-CLD-GCF* and *PUB-CLD-AWS* scaled better with the increase in user invocations as compared to other clusters. All these clusters were able to serve the user invocations with the consistent P90 response times. *PUB-CLD-AWS* cluster has the lowest P90 response times. Since both the cluster *PVT-CLD-LRZ-OW* and *PVT-CLD-LRZ-OF* have same resources, *PVT-CLD-LRZ-OW* could not perform well for this function. This shows the inability of OpenWhisk platform to work well as against the OpenFaaS platform for such functions. The edge cluster *Edge-Jetson-Nano* performed better than the *PVT-CLD-LRZ-OW*. The mean memory usage is almost constant across the two traces for all the clusters except the clusters based on OpenFaaS. For the *Edge-Jetson-Nano* cluster, it increased substantially with an increase in user invocations, while for the *PVT-CLD-LRZ-OF* cluster, it decreased since more function instances are created. The mean number of function instances created to handle the user invocations increased with `Trace R1` for all the clusters.

json-loads: Here, the *PUB-CLD-GCF* handled the highest number of invocation for `Trace 2`, but for `Trace 2`, the slowest cluster in `Trace 2`, i.e, the *PVT-CLD-LRZ-OW* cluster handled the highest number of invocations. The two public cloud clusters were able to serve the user invocations with the consistent P90 response times. Edge cluster is suited only for this function if the number of invocations are low. The mean memory usage is almost constant across the two traces for the two public cloud clusters, while for others, it increased

with an increase in user workload. It almost got doubled for the clusters based on OpenFaaS. Additionally, since this function involves downloading the file from the internet, we can see network transmission rise when load tested with `Trace R1` for all clusters.

lr-prediction: This function was only deployed on cloud clusters. For this function, the two public cloud clusters are a leap ahead of the private cloud clusters. Among the public cloud clusters, *PUB-CLD-GCF* always handled the highest number of invocations at a lower P90 response time than *PUB-CLD-AWS*. This function requires downloading the model from the storage bucket along with the data, and that process takes more than the set time limit for most invocations. As a result, both private cloud clusters could not handle the high workload for this function and hence were only able to serve a low number of successful invocations. Furthermore, this is not observed in the public cloud platforms, as they do model and data caching and could serve the invocations efficiently. The two public cloud clusters were able to serve the user invocations with consistent P90 response times. The mean number of function instances created to handle the user invocations increased with `Trace R1` for all public cloud clusters. For the *PVT-CLD-LRZ-OF* cluster, the mean number of function instances remained at one only, while for the *PVT-CLD-LRZ-OW* cluster, the mean number of function instances remains almost the same across two traces. This can be the result of them rejecting most of the invocations. Network usage by the *PUB-CLD-GCF* cluster is the lowest compared to other clusters, since the buckets to fetch the data and model reside within the Google cloud.

image-processing: For this function, the two public cloud clusters are again a leap ahead of the private cloud clusters. Among the public cloud clusters, *PUB-CLD-GCF* always handled the highest number of invocations at a lower P90 response time than *PUB-CLD-AWS*. This function requires downloading the image from the MinIO bucket, which is deployed on an instance of Google Cloud. Hence the *PUB-CLD-GCF* cluster has the lowest P90 response time. The mean memory usage remains almost the same across two traces for all the clusters. In this function, the OpenFaaS-based cluster performed much better than the OpenWhisk-based cluster, even though both have the same resources. This shows that the OpenWhisk-based platform is unsuitable for this type of function. The mean number of function instances created to handle the user invocations from `Trace R1` either increased or remained the same as with `Trace R2` for all the clusters.

10.3 FDN's Performance Overhead

FDN uses *Courier Load Balancer* (§6.2) based on HAProxy between the client and the clusters for load balancing. The clusters across which the invocations need to be delivered are automatically decided based on function delivery policies (§6.3.1). Also, the weights of clusters are decided based on different load-balancing algorithms (§6.3.2). Hence, a performance overhead can be induced by FDN compared to a direct function invocation. Therefore, we want to understand, *How much is the performance overhead introduced by the FDN's Courier Load Balancer when sending the invocations to the cluster compared to the direct invocation?*

For this evaluation, we selected **nodeinfo** (§9.1.1.1) function since it is a basic function without any overhead of its own and can run on all the clusters. The function is deployed on all clusters with `512MB` memory, and we send requests using the request traces (§9.3.2.3) to the function in two different scenarios:

- Direct connection (we perform the requests directly to the cluster endpoint). This scenario corresponds to the baseline.
- **Courier Load Balancer** (we perform the requests to the Courier endpoint)

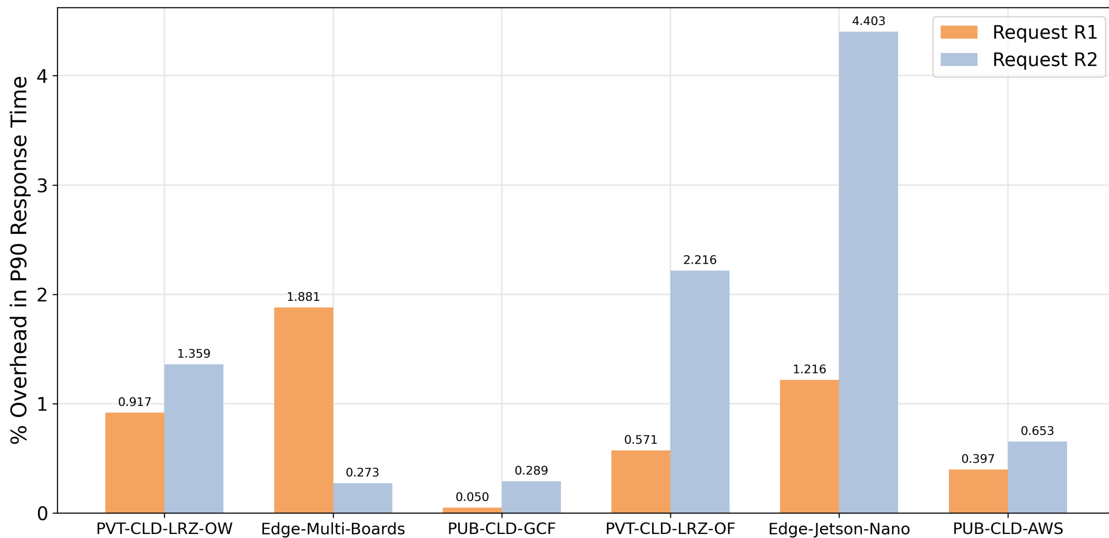


Figure 10.10.: Percentage overhead introduced in terms of P90 response time by FDN when compared against the direct approach for **nodeinfo** function across all the clusters and at two user workload invocations (R1 and R2). One can see that across all the clusters, the overhead is below 5%.

The evaluation is conducted for `20` minutes, and we compute the P90 response time of the requests in both scenarios. We calculate the percentage overhead introduced by the **Courier Load Balancer**. Figure 10.10 shows the percentage overhead introduced in P90 response time by FDN's **Courier Load Balancer** when compared against the direct approach for all the clusters and at two user workload invocations (R1 and R2). Overall, the overhead introduced by FDN can be attributed to the calculation needed by HAProxy to select a backend based on the function name. Across all the functions and two workloads, the overhead is below 5%. Therefore we conclude that the system overhead is negligible for the **Courier Load Balancer**.

10.4 FDN's Function Delivery Policies Correctness

FDN's *Courier Control Plane* creates various backends based on the function and data awareness. The policies based on these as described in §6.3.1 are called Function-Aware Delivery Policy (§6.3.1.1) and Data-aware Delivery Policy (§6.3.1.2) respectively. *Courier Control Plane* uses the function and data-awareness information to create various rules in the *Courier Load Balancer*. These rules then select a backend consisting of a subset of the available clusters and employ a load balancer for load balancing the incoming requests to the selected clusters. FDN also lets users override storage configurations and manually define where a master bucket should be replicated and which serverless compute clusters a bucket's load balancing route should contain. FDN allows users to control bucket replication. These policies include buckets' allowed clusters and their target replica counts. If a set of clusters has been defined, FDN will ensure that the bucket is only replicated to those clusters. Additionally, if a bucket's target replica count is altered, FDN will ensure that replica buckets are created and deleted where needed.

A scenario was devised to confirm the correctness of the set FDN's function delivery policies, wherein gradual changes are made using *FDN-UI* (§4.2.9), and the results of the *Courier Load Balancer* configuration are recorded. The steps followed are as follows:

1. We have used five clusters: `PUB-CLD-GCF`, `PUB-CLD-AWS`, `Edge-Jetson-Nano`, `PVT-CLD-LRZ-OW` and `PVT-CLD-LRZ-OF`. Four out of these clusters: `PUB-CLD-GCF`, `Edge-Jetson-Nano`, `PVT-CLD-LRZ-OW` and `PVT-CLD-LRZ-OF` belong to the zone *Germany*, while cluster `PUB-CLD-AWS` belongs to the zone *US*. Additionally, all cloud clusters belong to the zone *cloud*, while edge clusters belong to the zone *edge*. Since there are no buckets and functions deployed in the beginning, therefore there are no backends in the configuration file of *Courier Load Balancer* apart from the default backend:

Listing 10.1: Configuration file of *Courier Load Balancer* showing no backends apart from the default backend.

```
1 frontend http_front_courier
2    bind *:80
3    stats enable
4    stats uri /haproxy?stats
5    stats refresh 10s
6
7 backend default
8    balance roundrobin
9    server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
```

2. A bucket named *rep-policy-demo* is created, and an object named *image.png* is added to it. The bucket is set with the allowed zone *Germany* and the target replica count 0. It selects one of the cluster in the specified zone where the master bucket will reside. *Courier Load Balancer* configuration shows the bucket is only created on the `PVT-CLD-LRZ-OW` and therefore will direct function invocations requiring the bucket solely to the `PVT-CLD-LRZ-OW`:

Listing 10.2: Configuration file of *Courier Load Balancer* showing function invocations requiring the bucket *rep-policy-demo* solely go to the `PVT-CLD-LRZ-OW`.

```
1 frontend http_front_courier
2    bind *:80
3    stats enable
4    stats uri /haproxy?stats
5    stats refresh 10s
6    acl bucket_1 hdr(X-FDN-BUCKET) -i rep-policy-demo
7    use_backend bucket_1_policy if bucket_1
8
9 backend bucket_1_policy
10   balance roundrobin
11   server PVT-CLD-LRZ-OW 138.246.237.11:31003 weight 1
12
13 backend default
14    balance roundrobin
15    server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
```

3. The *rep-policy-demo* bucket's target replica count is changed from 0 to 2. *Courier Load Balancer* configuration shows the bucket has been replicated to the other two clusters (`PUB-CLD-GCF`, `PVT-CLD-LRZ-OF`), and the scheduling route now contains the three clusters:

Listing 10.3: Configuration file of *Courier Load Balancer* showing three clusters to which function invocations requiring the bucket *rep-policy-demo* solely go to, after increase in number of replica count of the bucket.

```
1 ...
2    acl bucket_demo hdr(X-FDN-BUCKET) -i demo
3    use_backend bucket_demo_policy if bucket_demo
4
5 backend bucket_demo_policy
6   balance roundrobin
7   server PVT-CLD-LRZ-OW 138.246.237.11:31003 weight 1
8   server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
9   server PUB-CLD-GCF 34.160.179.129:80 weight 1
10 backend default
11    balance roundrobin
12    server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
```

4. Now a test function *nodeinfo* is deployed on `PUB-CLD-AWS` and `PVT-CLD-LRZ-OF`. *Courier Load Balancer* configuration shows two more backends are created, one for the function only and one matching both function and bucket:

Listing 10.4: Configuration file of *Courier Load Balancer* showing two additional backends after a test function *nodeinfo* is deployed; one for the function and one matching both function and bucket

```
1 ...
2    acl bucket_demo hdr(X-FDN-BUCKET) -i demo
3    use_backend bucket_demo_policy if bucket_demo
4
5    acl url_func_nodeinfo path_beg /function/nodeinfo
6    use_backend function_nodeinfo_policy if url_func_nodeinfo
7
8    use_backend nodeinfo_bucket_demo_policy if url_func_nodeinfo AND bucket_demo
9
10 backend nodeinfo_bucket_demo_policy
11   balance roundrobin
12   server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
13
14 backend function_nodeinfo_policy
15   balance roundrobin
16   server PUB-CLD-AWS 44.209.191.63:80 weight 1
17   server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
18
19 backend bucket_demo_policy
20   balance roundrobin
21   server PVT-CLD-LRZ-OW 138.246.237.11:31003 weight 1
22   server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
23   server PUB-CLD-GCF 34.160.179.129:80 weight 1
24 backend default
25    balance roundrobin
26    server PVT-CLD-LRZ-OF 138.246.236.155:31112 weight 1
```

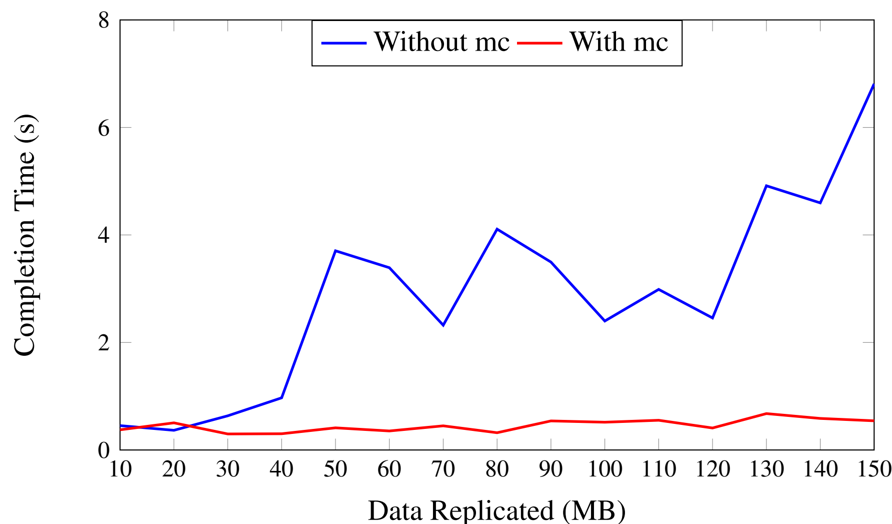


Figure 10.11.: Mirroring different data amounts from the PVT-CLD-LRZ-OF to PVT-CLD-LRZ-OW using mc and without it.

These observations confirm the correctness of FDN's function delivery policies. The FDN ensures that buckets are only replicated to allowed locations, reacting to changes to keep the replicated data up-to-date and the replica buckets consistent with user set policies.

10.5 FDN's Bucket Replication Performance

FDN's bucket replication performance depends entirely on MinIO's mc command-line tool and its mc mirror directive. It is used for replicating the MinIO buckets across clusters based on the target replica counts specified by the user. Therefore, to assess FDN's bucket replication performance, it was useful to isolate the command-line tool and measure its performance directly. To accomplish this, a bucket was created on the PVT-CLD-LRZ-OF with its replica in PVT-CLD-LRZ-OW and the performance of replication of different data amounts ranging from 10MB to 150MB is analyzed under the following two scenarios:

1. Use mc mirror for copying the primary bucket data to the mirrored bucket every time a change is made (With mc in Figure 10.11)
2. Manually full copying the primary bucket data to the mirrored bucket every time a change is made (Without mc in Figure 10.11).

It is to be noted that the new data is cumulatively added to the existing data. Performance variation for the two scenarios can be seen in Figure 10.11. In the first scenario, the secondary bucket contains some of the primary's content and only receives the new data at each step. This is because MinIO's mirroring tool works similarly to Andrew Tridgell and Paul Mackerras' *rsync* utility [197, 294], where the tool calculates the difference between the two buckets and only transfers the data necessary to make their content match. While in the second scenario, the whole bucket data was copied and took much longer. It is a significant advantage to FDN, as it dramatically reduces the amount of data FDN must transfer for replication tasks, which is helpful in case the master bucket resides on an edge cluster and the secondary bucket in the cloud. It also protects FDN from many large replication jobs that mutex lock the mc tool for long periods.

10.6 FDN's Load Balancing Algorithms Performance

In order to distribute the load of the incoming invocations among the target serverless compute clusters spread across the edge-cloud continuum, FDN uses *Courier Load Balancer* (§6) that sits between them and the user. The *Courier Load Balancer* receives the user's requests, and the requests are dispatched to the second layer using set function delivery policies (§6.3.1) where, depending on the policy, a load balancer is employed. Using the second layer load balancer, invocations are load balanced across the subset of serverless compute clusters based on different load balancing algorithms (§6.3.2). We analyze the performance of the following load balancing algorithms within FDN:

1. **FDN Round-Robin Algorithm (FDN-RR)**: This algorithm simply distributes the user invocations equally to all available target clusters. It may have the disadvantage that it does not directly react to changes in the runtime behavior of functions or the load on the target cluster [104].

2. **FDN Latency-Aware Algorithm (FDN-Latency-Aware)**: This approach adapts the weights of clusters according to the functions' execution time in the target clusters (§6.3.2.1). This is done to reflect the latency for each function invocation within the clusters and automatically take into account the computational capability of the cluster and available free resources.

3. **FDN SLO-Aware Algorithm (FDN-SLO-Aware)**: This approach adapts the weights of clusters according to the functions' execution time in the target clusters (§6.3.2.2) and defined SLOs. If the execution time on a cluster goes beyond the defined SLO, then the weight of the cluster will be defined as zero. Otherwise, it is calculated in the same fashion as in the *FDN-Latency-Aware Load Balancing Algorithm.*

4. **FDN Least Connections Algorithm (FDN-LeastCon)**: It keeps track of the number of open connections to a target cluster from the Courier Load Balancer. It distributes invocations to the cluster with the fewest open connections.

5. **FDN Round-Robin Algorithm Cloud Only (FDN-RR-Cld)**: This approach is the same as *Round-Robin Algorithm*; however, it only distributes invocations across cloud clusters.

6. **FDN Latency-Aware Algorithm Cloud Only (FDN-Latency-Aware-Cld)**: This approach is the same as *FDN-Latency-Aware Algorithm*; however, as the name suggests, it only distributes invocations across cloud clusters.

7. **FDN SLO-Aware Algorithm Cloud Only (FDN-SLO-Aware-Cld)**: This approach is the same as *SLO-Aware Algorithm*; however, as the name suggests, it only distributes invocations across cloud clusters.

8. **FDN Least Connections Algorithm Cloud Only (FDN-LeastCon-Cld)**: This algorithm is the same as *FDN-LeastCon Algorithm*, but it only distributes invocations across cloud clusters.

To confirm FDN's load balancing performance, measuring and recording function execution times under various scenarios is necessary. These measurements will provide insight into FDN's overall performance and the impact of different load-balancing algorithms. This section presents the result of FDN's load balancing across multiple clusters spread across the edge-cloud continuum. We focus on the following aspects:

- **Performance on Individual FaaS functions**: Here, we focus the performance of the load balancing algorithms on individual FaaS functions and try to answer the question: *How does the different FDN's load balancing perform against each other, and which algorithm are better suited for which functions?*

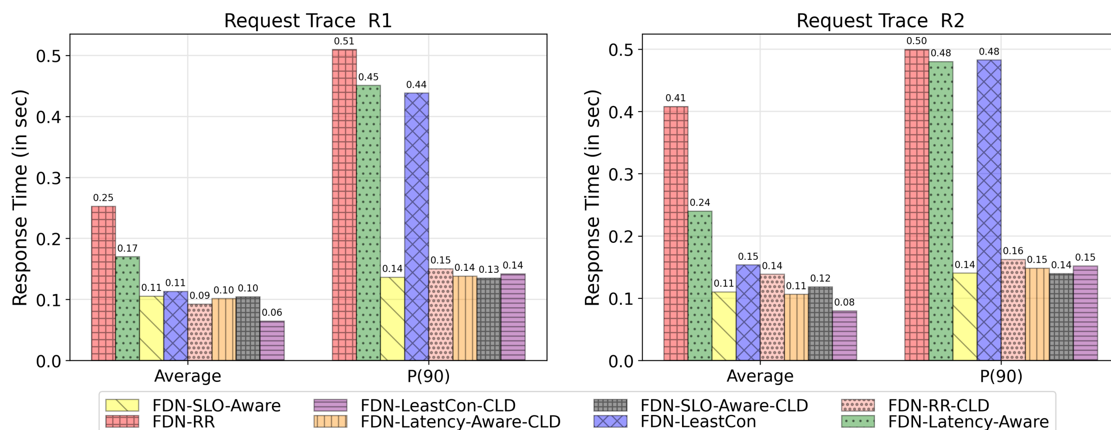


Figure 10.12.: The average and 90th percentile response times of the invocations load balanced using eight different algorithms to the `nodeinfo` function. The results are shown for both *Invocation Traces*.

- **Performance on Serverless application**: Here the focus is on the performance of the serverless application. The individual functions in the application are load balanced with different algorithms. The combined effect of the load balancing of individual functions will be seen in the application's performance.
- **Performance on high user workload invocations**: Here, we measure the load balancing performance of FaaS Functions and the serverless application on high user workload invocations.

Each evaluation was conducted for `20` minutes using the two *Invocation Traces* (R1 and R2, §9.3.2.3). Since most of the results related to load balancing are similar, therefore, we focus our results and analysis of the algorithms in the next subsections on **three benchmark functions** from different categories.

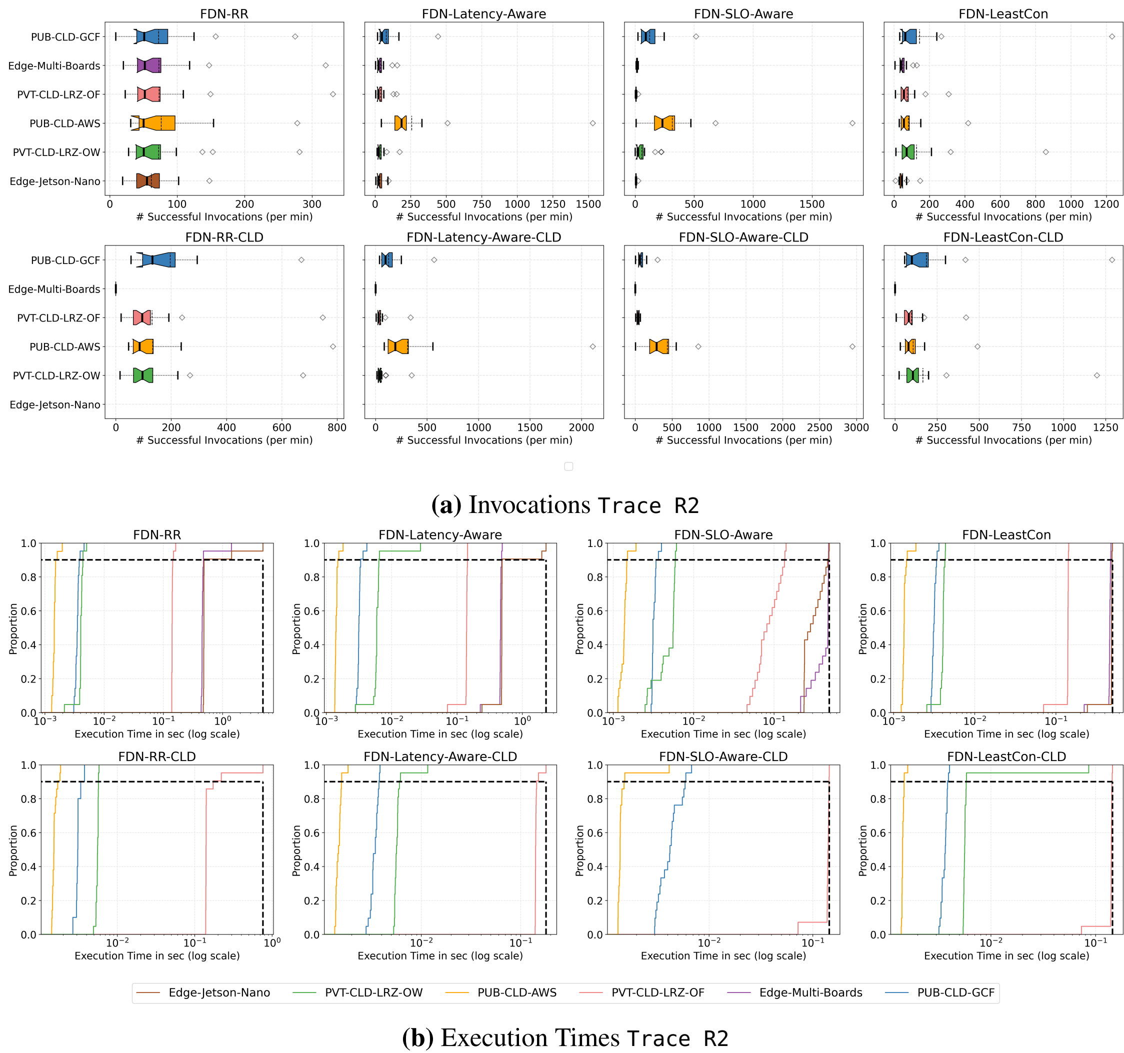
10.6.1 Individual FaaS Function (nodeinfo)

Here, we evaluate the load balancing performance of FDN on `nodeinfo` function.

10.6.1.1 Performance on Low Workload (Trace R2)

The `nodeinfo` function was deployed on all the clusters (§9.2). FDN-RR, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon algorithms load balance the user invocation request to all the clusters, while FDN-Latency-Aware-Cld, FDN-SLO-Aware-Cld and FDN-LeastCon-Cld load balance only across the cloud clusters. Figure 10.12 shows the summarized results of the evaluation, where we show the average and 90th percentile response times of the invocations to the `nodeinfo` function load balanced using eight different algorithms.

From Figure 10.12 for `Trace R2`, we observe that FDN-RR has the highest P90 response time of `0.50s`. It is followed by FDN-Latency-Aware and FDN-LeastCon with P90 response times of `0.48s`. FDN-SLO-Aware has the lowest P90 response time of `0.14s`. Among the cloud-only algorithms, we observe that again FDN-RR-Cld has the highest P90 response time of `0.16s`. It is followed by FDN-Latency-Aware-Cld and FDN-LeastCon-Cld with P90 response time of `0.15s`. Again, FDN-SLO-Aware-Cld has the lowest P90 response time of `0.14s`. Across all the algorithms, FDN-SLO-Aware and its cloud version performed the



(a) Invocations `Trace R2`

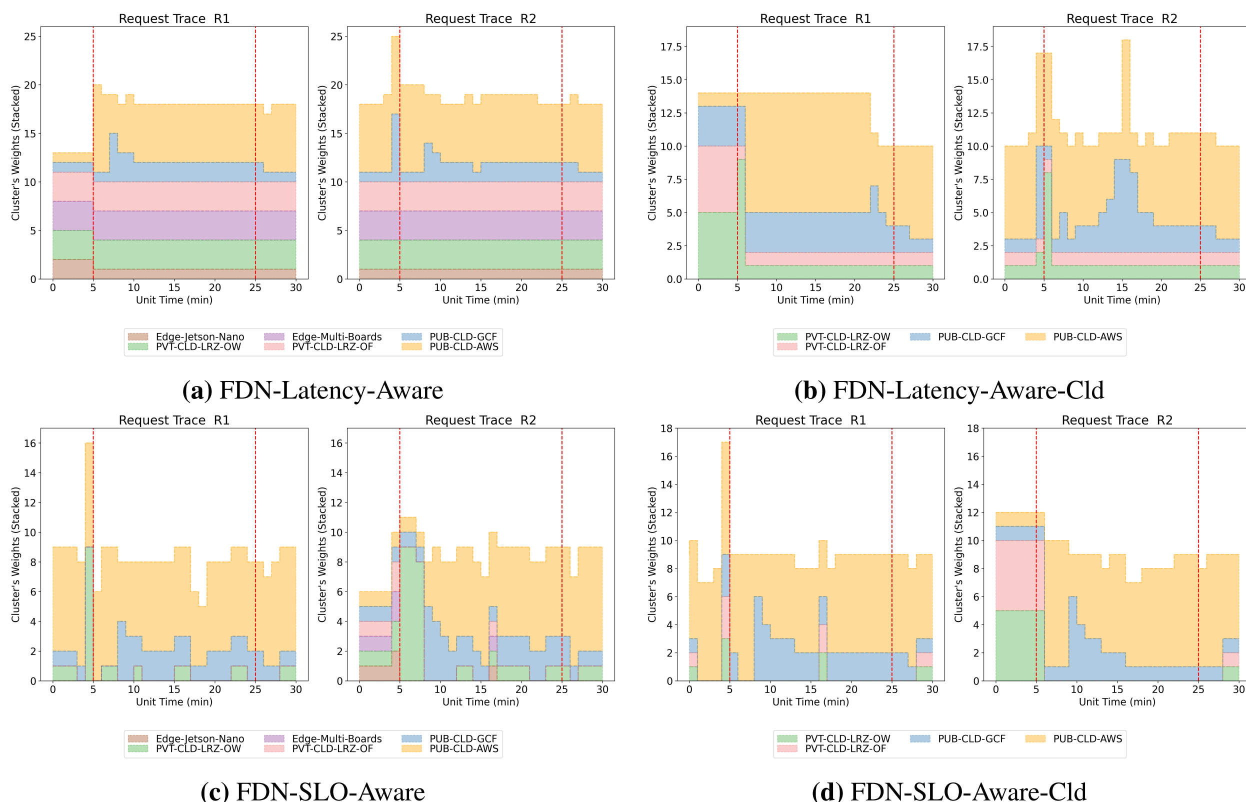
(b) Execution Times `Trace R2`

Figure 10.13.: Details on how the successful invocations made using `Trace R2` to `nodeinfo` function are distributed across each cluster along with their execution times using different load balancing algorithms.

best for `Trace R2`. Figure 10.13 shows details on how the successful invocations made using `Trace R2` to `nodeinfo` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

It is to be noted that, in the case of measuring the performance of load balancing algorithms, we use MET. In contrast, for measuring the performance of the individual functions in §9.1.1 we used P90 response time. Since our designed algorithms and load balancer are on the server side, and they use the monitoring data collected from the clusters (execution duration of the functions) to make the decisions, we show performance in terms of MET. Thus, this performance metric only considers the current execution time on the cluster and does not consider the cluster's location and the latency impact it might have on the client side. However, this shortcoming of the current load balancing algorithms can be improved in the future by taking into account the latency between the cluster location and the client. The performance of the load balancing algorithms is presented below:

FDN-RR: Each cluster has almost successfully handled the same number of invocations, approximately `65`



(a) FDN-Latency-Aware

(b) FDN-Latency-Aware-Cld

(c) FDN-SLO-Aware

(d) FDN-SLO-Aware-Cld

Figure 10.14.: Weights distribution among different clusters during evaluation test for `nodeinfo` function when load balanced using different algorithms for two *Invocation Traces*.

MSI/min to `75` MSI/min. The execution time of invocations on each cluster varies. The *PUB-CLD-AWS* cluster has the lowest MET of `0.0014s`, followed by the *PUB-CLD-GCF* cluster with MET of `0.0035s`, then comes the two private cloud clusters. The MET of invocations on the *PVT-CLD-LRZ-OW* cluster is `0.004s`, while on the *PVT-CLD-LRZ-OF* cluster is `0.142s`. The two edge clusters have the highest MET. The MET of invocations on the *Edge-Multi-Boards* cluster is `0.50s`, while on the *Edge-Jetson-Nano* cluster is `0.73s`. The slowest cluster *Edge-Jetson-Nano* impacted the performance of collaborative execution among the clusters, and that's why the P90 response time for `Trace R2` for FDN-RR is `0.50s` (see Figure 10.12).

FDN-Latency-Aware : When using this algorithm, most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `257.47` MSI/min, since it has the lowest MET of `0.001s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `78.42` MSI/min with MET of `0.0032s`. Other clusters handled around `35` MSI/min, as their execution times are higher than the two public cloud clusters. The weight assignment across clusters during the evaluation can be observed in Figure 10.14a. We can observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster. The clusters *PVT-CLD-LRZ-OF*, *PVT-CLD-LRZ-OW* and *Edge-Multi-Boards* clusters are assigned the weight of three for the entire evaluation, while the cluster *Edge-Jetson-Nano* is assigned a weight of one. The *PUB-CLD-GCF* cluster weight varied from one to four, but was kept at two for most of the evaluation period. Overall P90 response time using this algorithm is `0.48s`. This algorithm gave the advantage that the two edge clusters can also contribute in handling the invocations alongside the cloud clusters.

FDN-SLO-Aware: In this algorithm, we have set the SLO as `1s`. Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `314.66` MSI/min with MET of `0.0013s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `122.45` MSI/min with MET of `0.003s`. The *PVT-CLD-LRZ-OW* cluster

handled 55.46 MSI/min with MET of 0.0048s. The *Edge-Jetson-Nano* and *PVT-CLD-LRZ-OF* clusters handled only around 7 MSI/min with MET of 0.089s on *PVT-CLD-LRZ-OF* and 0.32s on *Edge-Jetson-Nano*. The *Edge-Multi-Boards* cluster handled 15.67 MSI/min, with MET of 0.410s. It can also be evident from Figure 10.14c, where we show the weights' distribution among different clusters (in stacked format). For Trace R2, we can observe that initially, the highest weights are assigned to the *PVT-CLD-LRZ-OW* cluster. Before this timestamp, we can observe that all the clusters are assigned almost equal weights. This is due to the algorithm not having any execution time of the function on each cluster; therefore, it distributes the maximum sum weight (here 10) across each cluster. Afterward, based on the execution time results, the highest weights are assigned to the *PVT-CLD-LRZ-OW* cluster, followed by *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters. All the other clusters are assigned a zero weight since their initial cold-start execution time was greater than one second. Afterward, due to high execution time, the algorithm also set a weight of zero for the *PVT-CLD-LRZ-OW* cluster. Over the evaluation time, we can observe that the *PUB-CLD-AWS* cluster is assigned with the highest weights, followed by the *PUB-CLD-GCF* cluster. Overall P90 response time with this algorithm is 0.14s.

FDN-LeastCon: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled 144.14 MSI/min with MET of 0.0031s. It is closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled 126.6 MSI/min with MET of 0.004s. Then comes the *PUB-CLD-AWS* cluster, which handled 81.23 MSI/min with MET of 0.0013s. The *PVT-CLD-LRZ-OF* cluster handled 74.33 MSI/min with MET of 0.13s. Both edge clusters handled around 50 MSI/min. FDN-LeastCon algorithm following this distribution resulted in an overall P90 response time of 0.48s.

FDN-RR-Cld: Here each cloud cluster has almost successfully handled the same number of invocations, approximately 133.1 MSI/min except the *PUB-CLD-GCF* cluster, which handled 196.4 MSI/min. We do not have concrete reasoning for this behavior, but we assume that the other clusters returned errors for the rest of the invocations. The *PUB-CLD-AWS* cluster has the lowest MET of 0.0015s. The *PUB-CLD-GCF* cluster has MET of 0.0031s, the *PVT-CLD-LRZ-OW* cluster has MET of 0.005s and *PVT-CLD-LRZ-OF* cluster has MET of 0.175s. The slowest cluster, i.e., *PVT-CLD-LRZ-OF*, impacted the performance of collaborative execution among the clusters. The P90 response time for Trace R2 for FDN-RR-Cld is 0.16s (see Figure 10.12).

FDN-Latency-Aware-Cld: In this algorithm, we can again observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 311.2 MSI/min with MET of 0.0014s. It is followed by the *PUB-CLD-GCF* cluster, which handled 128.28 MSI/min with MET of 0.0034s. Both private cloud clusters handled approximately 55 MSI/min since their execution time is high. The *PVT-CLD-LRZ-OW* cluster has MET of 0.0059s, while *PVT-CLD-LRZ-OF* cluster has MET of 0.143s. This can also be evident from Figure 10.14b, where we see the weights' distribution among different clusters for Trace R2. we observe that, initially, higher weights were assigned to the *PVT-CLD-LRZ-OW* and *PUB-CLD-AWS* clusters. However, in the next minute, the weight for the *PVT-CLD-LRZ-OW* cluster is set to one. The highest weights throughout the evaluation period are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster. We can observe a lower overall P90 response time of 0.15s (see Figure 10.12).

FDN-SLO-Aware-Cld: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 435.47 MSI/min with MET of 0.0015s. It is followed by the *PUB-CLD-GCF* cluster, which handled 82.76 MSI/min with MET of 0.0043s. Among private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled approximately 5 MSI/min with MET of 0.13s, while the *PVT-CLD-LRZ-OW* cluster did not handle any invocation. It can be attributed to the initial high cold start time in OpenWhisk. As a result, this algorithm sets its weight to zero at the beginning of the evaluation. This can also be evident from Figure 10.14d, where we see the weights' distribution among different clusters for Trace R2. The highest weights throughout the evaluation period are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster. The

other clusters are assigned zero weight. We can observe the overall P90 response time of `0.14s` same as that of FDN-SLO-Aware.

FDN-LeastCon-Cld: In this algorithm, most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `186.90` MSI/min with MET of `0.0036s`. It is closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled `165.76` MSI/min with MET of `0.0094s`. Then comes the *PUB-CLD-AWS* cluster, which handled `107.76` MSI/min with lowest MET of `0.0014s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled `98.7` MSI/min with MET of `0.138s`. The overall P90 response time using this algorithm is `0.15s` (see Figure 10.12).

10.6.1.2 Performance on High Workload (Trace R1)

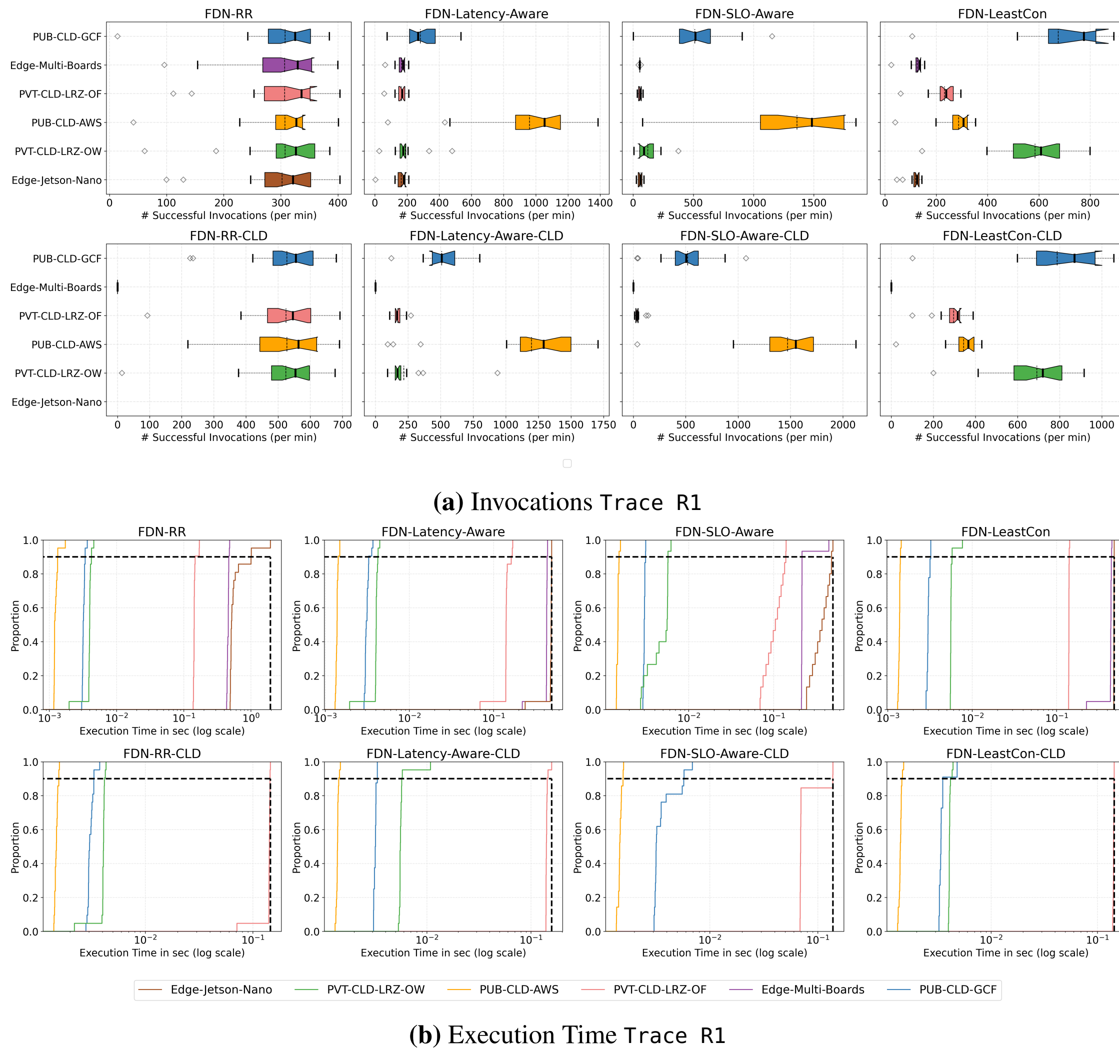
From Figure 10.12 for `Trace R1`, we observe that FDN-RR has the highest P90 response time of `0.51s`. It is followed by FDN-Latency-Aware with P90 response time of `0.45s`, then FDN-LeastCon with P90 response time of `0.44s`. FDN-SLO-Aware has the lowest P90 response time of `0.14s`. Among the cloud-only algorithms, we observe that again FDN-RR-Cld has the highest P90 response time of `0.15s`. It is followed by FDN-Latency-Aware-Cld and FDN-LeastCon-Cld with P90 response time of `0.14s`. Again, FDN-SLO-Aware-Cld has the lowest P90 response time of `0.13s`. Across all the algorithms, again, FDN-SLO-Aware and its cloud version performed the best for `Trace R1`.

Figure 10.15 shows details on how the successful invocations made using `Trace R1` to `nodeinfo` function are distributed across each cluster, along with their execution times using different load balancing algorithms. The performance of the load balancing algorithms is presented below:

FDN-RR: Each cluster has successfully handled the same number of invocations, approximately `307` MSI/min. However, the response time on each cluster differs to a great extent. MET on the *Edge-Jetson-Nano* cluster is `0.63s`, `0.456s` for the *Edge-Multi-Boards*, `0.004s` for the *PVT-CLD-LRZ-OW*, `0.141s` for the *PVT-CLD-LRZ-OF*, `0.0012s` for the *PUB-CLD-AWS*, and `0.003s` for the *PUB-CLD-GCF*. The lowest execution time is on the *PUB-CLD-AWS* cluster. The slowest cluster, i.e., *Edge-Jetson-Nano* has impacted the performance of collaborative execution among the clusters and that's why in Figure 10.12, we saw the P90 response time for `Trace R1` for FDN-RR as `0.51s`.

FDN-Latency-Aware: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `960.33` MSI/min, since it has the lowest MET of `0.0013s`) as compared to the other clusters. It is followed by the *PUB-CLD-GCF* cluster, which handled `282.2` MSI/min with MET of `0.0031s`. Other clusters handled lower number of invocations. Among edge clusters, *Edge-Jetson-Nano* handled `163.7` MSI/min with MET of `0.47s`, while the *Edge-Multi-Boards* cluster handled `166.23` MSI/min with MET of `0.420s`. Among private clusters, the *PVT-CLD-LRZ-OW* cluster handled `189.10` with MET of `0.0039s`, while *PVT-CLD-LRZ-OF* cluster handled `165.48` with MET of `0.14s`. These invocations across clusters are decided upon the weights assigned to the clusters. In Figure 10.14a, we show the weights' distribution among different clusters (in stacked format). For `Trace R1`, we can observe that the highest weights during the evaluation phase of the test are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster. The clusters *PVT-CLD-LRZ-OF*, *PVT-CLD-LRZ-OW* and *Edge-Multi-Boards* clusters are assigned the weight of three for the entire evaluation, while the cluster *Edge-Jetson-Nano* is assigned a weight of one. The *PUB-CLD-GCF* cluster weight varied from one to four, but was at two for most of the evaluation period.

FDN-SLO-Aware: In this algorithm, we have set the SLO as `1s`. Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `1360.20` MSI/min, since it has the lowest MET of `0.0014s`) as compared to the other clusters. It is followed by the *PUB-CLD-GCF* cluster, which handled `512.13` MSI/min with MET of `0.003s`. This algorithm follows a similar trend as FDN-Latency-Aware since the execution



(a) Invocations `Trace R1`

(b) Execution Time `Trace R1`

Figure 10.15.: Details on how the successful invocations made using `Trace R1` to `nodeinfo` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

time does not go beyond the defined SLO. The *PVT-CLD-LRZ-OW* cluster handled `121.745` MSI/min with MET of `0.004s`. This algorithm did not send many requests to the slow clusters. As a result, all the private cloud clusters and the edge clusters, handled `59` MSI/min. It can also be evident from Figure 10.14c, where we show the weights' distribution among different clusters (in stacked format). For `Trace R1`, we can observe that the weights are distributed only across the *PUB-CLD-AWS*, *PUB-CLD-GCF* and *PVT-CLD-LRZ-OW* clusters. Overall P90 response time with this algorithm is `0.14s` (see Figure 10.12).

FDN-LeastCon: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `675.33` MSI/min with MET of `0.003s`. It is closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled `584.88` MSI/min with MET of `0.0056s`. Then comes the *PUB-CLD-AWS* cluster, which handled `87.04` MSI/min with MET of `0.0013s` and the *PVT-CLD-LRZ-OF* cluster, which handled `231.8` MSI/min with MET of `0.14s`. Both edge clusters handled the lowest number of invocations. The *Edge-Jetson-Nano* cluster handled `119.4` MSI/min with MET of `0.478s`, while the *Edge-Multi-Boards* cluster handled `125.4` MSI/min with MET of `0.430s`. FDN-LeastCon algorithm following this distribution resulted in an overall

P90 response time of 0.44s.

FDN-RR-Cld: Each cluster here has almost successfully handled the same number of invocations, approximately, 525.0 MSI/min. However, the response time on each cluster differs to a great extent. MET on the *PVT-CLD-LRZ-OW* cluster is 0.0040s, 0.139s for the *PVT-CLD-LRZ-OF*, 0.0014s for the *PUB-CLD-AWS*, and 0.003s for the *PUB-CLD-GCF*. The lowest execution time is on the*PUB-CLD-AWS* cluster. From Figure 10.12, we observe that the P90 response time for Trace R1 for FDN-RR-Cld is 0.15s.

FDN-Latency-Aware-Cld: We can observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 1196.57 MSI/min, since it has the lowest MET of 0.0013s as compared to the other clusters. It is followed by the *PUB-CLD-GCF* cluster, which handled 508.87 MSI/min with MET of 0.0031s. Private cloud clusters handled lower number of invocations, 218.23 MSI/min by the *PVT-CLD-LRZ-OW* cluster with MET of 0.0058s, while 171.66 MSI/min by the *PVT-CLD-LRZ-OF* cluster with MET of 0.14s. The number of invocations by a cluster is decided based on the execution time of the function on that cluster, which can also be evident from Figure 10.14b, where we show the weights' distribution among different clusters. We can observe for Trace R1, the highest weights during the evaluation phase assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster, since they have the lowest MET. The weights on other clusters remained zero for the entire evaluation. Since this algorithm does not distribute invocations to edge clusters, We can observe a lower overall P90 response time of 0.14s (see Figure 10.12).

FDN-SLO-Aware-Cld: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 1471.5 MSI/min with MET of 0.0014s. It is followed by the *PUB-CLD-GCF* cluster, which handled 517.88 MSI/min with MET of 0.0037s. Among private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled 46.15 MSI/min with MET of 0.0798s, while the *PVT-CLD-LRZ-OW* cluster did not handle any invocation. It can be attributed again to the high cold start time in OpenWhisk. As a result, the weight of this cluster is set to zero. This can also be evident from Figure 10.14d, where we see the weights' distribution among different clusters for Trace R1. The highest weights throughout the evaluation period are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster. We can observe the overall P90 response time of 0.14s same as that of FDN-SLO-Aware.

FDN-LeastCon-Cld: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled 788.09 MSI/min with MET of 0.0034s. It is closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled 691.27 MSI/min with MET of 0.004s. Then comes the *PUB-CLD-AWS* cluster, which handled 343.19 MSI/min with MET of 0.0014s and lastly the *PVT-CLD-LRZ-OF* cluster, which handled 295.89 MSI/min with MET of 0.142s. FDN-LeastCon-Cld algorithm following this distribution resulted in an overall P90 response time of 0.14s.

10.6.2 Individual FaaS Function (gzip-compression)

The gzip-compressionfunction was deployed on all the clusters (§9.2) except *Edge-Multi-Boards* because of it's low compute capabilities. Figure 10.16 shows the summarized results of the evaluation, where we show the average and 90th percentile of the response times of the invocation requests, load balanced using eight different algorithms.

10.6.2.1 Performance on Low Workload (Trace R2)

From Figure 10.16 for Trace R2, we observe that across the two metrics, FDN-RR has the highest P90 response time of 1.66s. It is followed by FDN-Latency-Aware with P90 response time of 1.55s, then FDN-LeastCon with P90 response time of1.42s. FDN-SLO-Aware has the lowest P90 response time of

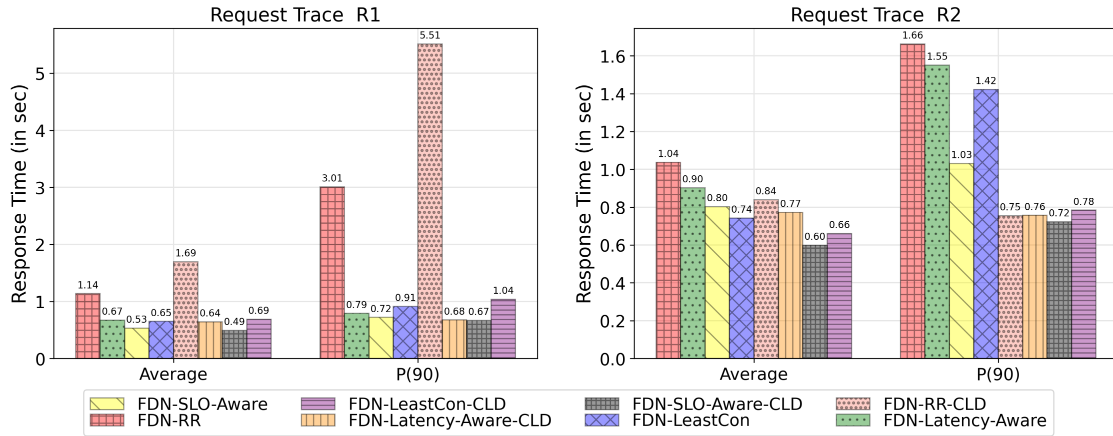


Figure 10.16.: The average and 90th percentile of the response times of the invocation requests, load balanced using eight different algorithms to the `gzip-compression` function. The results are shown for two *Invocation Traces*.

`1.02s`. Among the cloud-only algorithms, we observe that across the two metrics, FDN-RR-Cld has the P90 response time of `0.75s`, FDN-Latency-Aware-Cld has P90 response time of `0.76s`, FDN-LeastCon-Cld has P90 response time of `0.78s`, and FDN-SLO-Aware-Cld has P90 response time of `0.72s`. Again, FDN-SLO-Aware-Cld has the lowest P90 response time across all the algorithms for `Trace R2`.

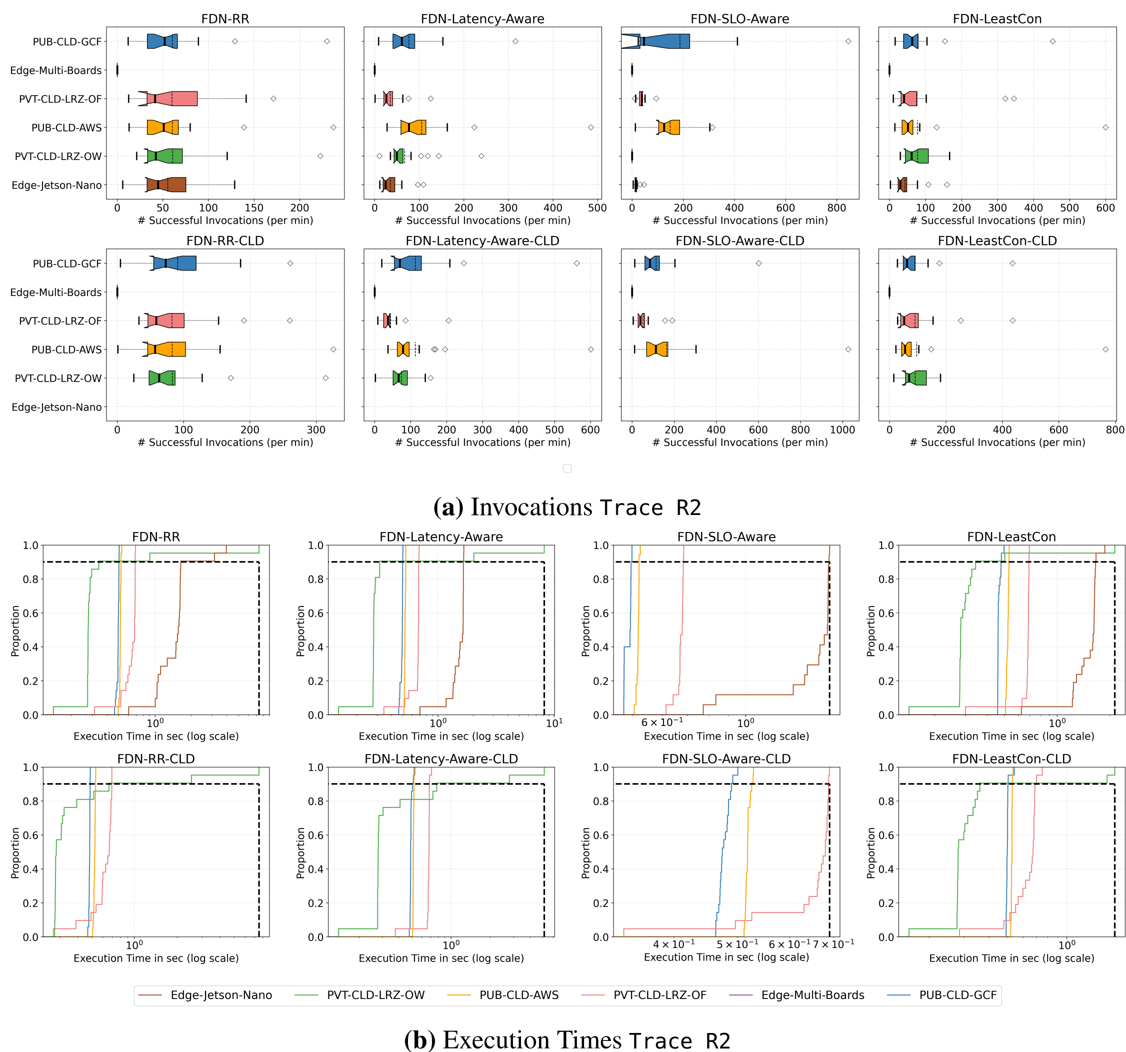
Figure 10.17b shows details on how the successful invocations made using `Trace R2` are distributed across each cluster, along with their execution times using different load balancing algorithms.

Performance of the load balancing algorithms is presented below:

FDN-RR: Each cluster has almost handled `60` MSI/min. The *PUB-CLD-GCF* cluster has the lowest execution time with MET of `0.491s`. It is followed by the *PUB-CLD-AWS* cluster with MET of `0.513s`. Both private clusters have MET of `0.63s`. The slowest cluster, in this case is *Edge-Jetson-Nano* with MET of `1.59s` has impacted the performance of collaborative execution among the clusters, and that's why the P90 response time for `Trace R2` is `1.66s` (see Figure 10.16).

FDN-Latency-Aware: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `105.5` MSI/min with MET of `0.521s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `76.85` MSI/min with MET of `0.48s`. Among private cloud clusters, the *PVT-CLD-LRZ-OW* cluster handled `67.50` MSI/min with MET of `0.73s`, while the *PVT-CLD-LRZ-OF* cluster handled `35.16` MSI/min with MET of `0.64s`. Edge cluster *Edge-Jetson-Nano* handled `35.0` MSI/min with MET of `1.50s`. The weight assignment across clusters during the evaluation can be observed in Figure 10.18a. We can observe, for `Trace R2`, that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* and *PVT-CLD-LRZ-OW* clusters. Overall P90 response time by using this algorithm is `1.55s` (see Figure 10.16).

FDN-SLO-Aware: The SLO was set as `1s`. Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `187.2` MSI/min at MET of `0.48s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `148.8` MSI/min at MET of `0.51s`. Among private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled `37.28` MSI/min at MET of `0.67s`, while no invocation was sent to the *PVT-CLD-LRZ-OW* cluster due to its high initial cold-start time. Edge cluster *Edge-Jetson-Nano* only handled `16.9` MSI/min with MET of `1.5s`. From Figure 10.18c, we can observe the weights' distribution among different clusters



(a) Invocations `Trace R2`

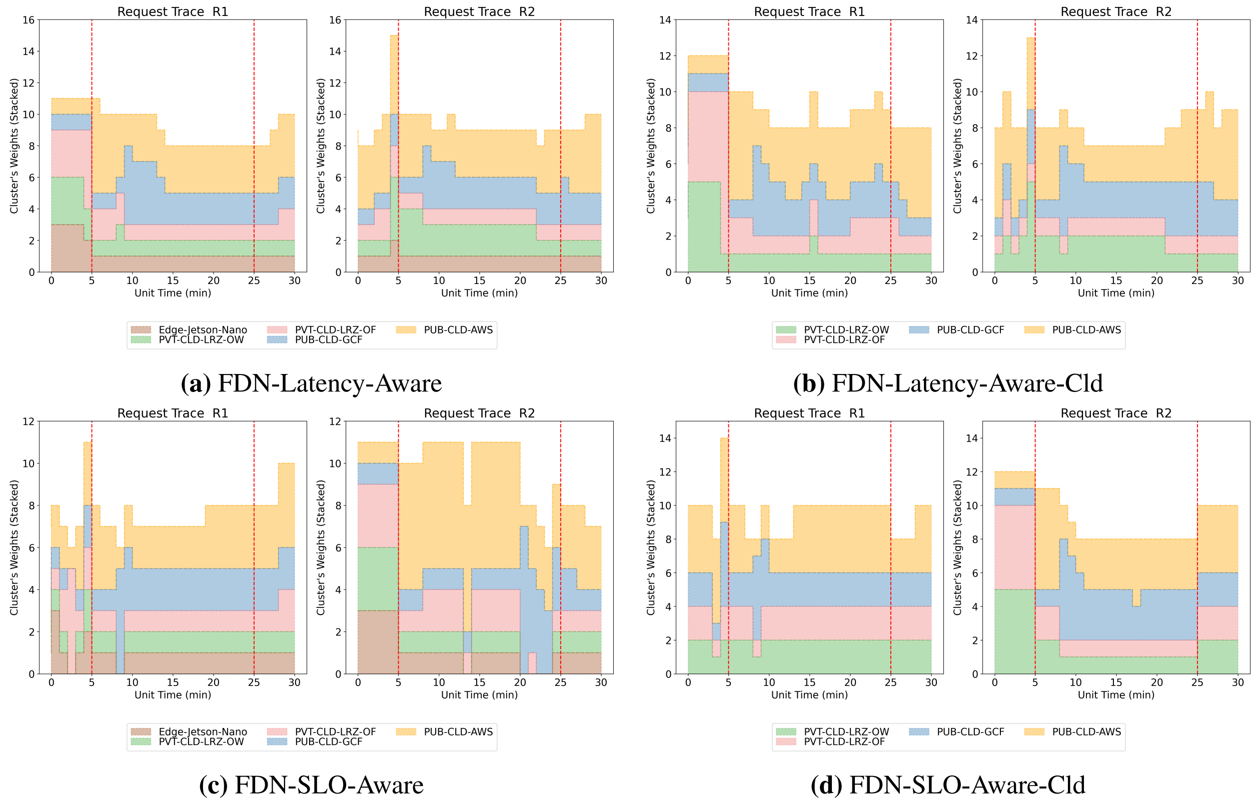
(b) Execution Times `Trace R2`

Figure 10.17.: Details on how the successful invocations made using `Trace R2` to `gzip-compression` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

(in stacked format). For `Trace R2`, the highest weights are assigned to the *PUB-CLD-AWS* and *PUB-CLD-GCF* cluster. Overall P90 response time with this algorithm is `1.03s`, which is around the set SLO of one second.

FDN-LeastCon: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `79.80` MSI/min with MET of `0.46s`. It is closely followed by the *PVT-CLD-LRZ-OW* cluster, which handled `78.0` MSI/min with MET of `0.38s`, and also the *PUB-CLD-AWS* cluster, which handled `77.76` MSI/min with MET of `0.52s`. Private cloud cluster *PVT-CLD-LRZ-OF* handled `75.22` MSI/min with MET of `0.66s`. Lastly, edge cluster *Edge-Jetson-Nano* handled `43.2` MSI/min with MET of `1.50s`. FDN-LeastCon algorithm resulted in an overall P90 response time of `1.42s` (see Figure 10.16).

FDN-RR-Cld: Again, each cloud cluster has almost successfully handled the same number of invocations, approximately `85` MSI/min. The *PUB-CLD-GCF* cluster has the lowest MET of `0.48s`. The slowest cluster, in this case, is the *PVT-CLD-LRZ-OW* cluster with MET of `0.77s`. It has impacted the performance of collaborative execution among the clusters. The P90 response time for `Trace R2` for FDN-RR-Cld is `0.75s`



(a) FDN-Latency-Aware

(b) FDN-Latency-Aware-Cld

(c) FDN-SLO-Aware

(d) FDN-SLO-Aware-Cld

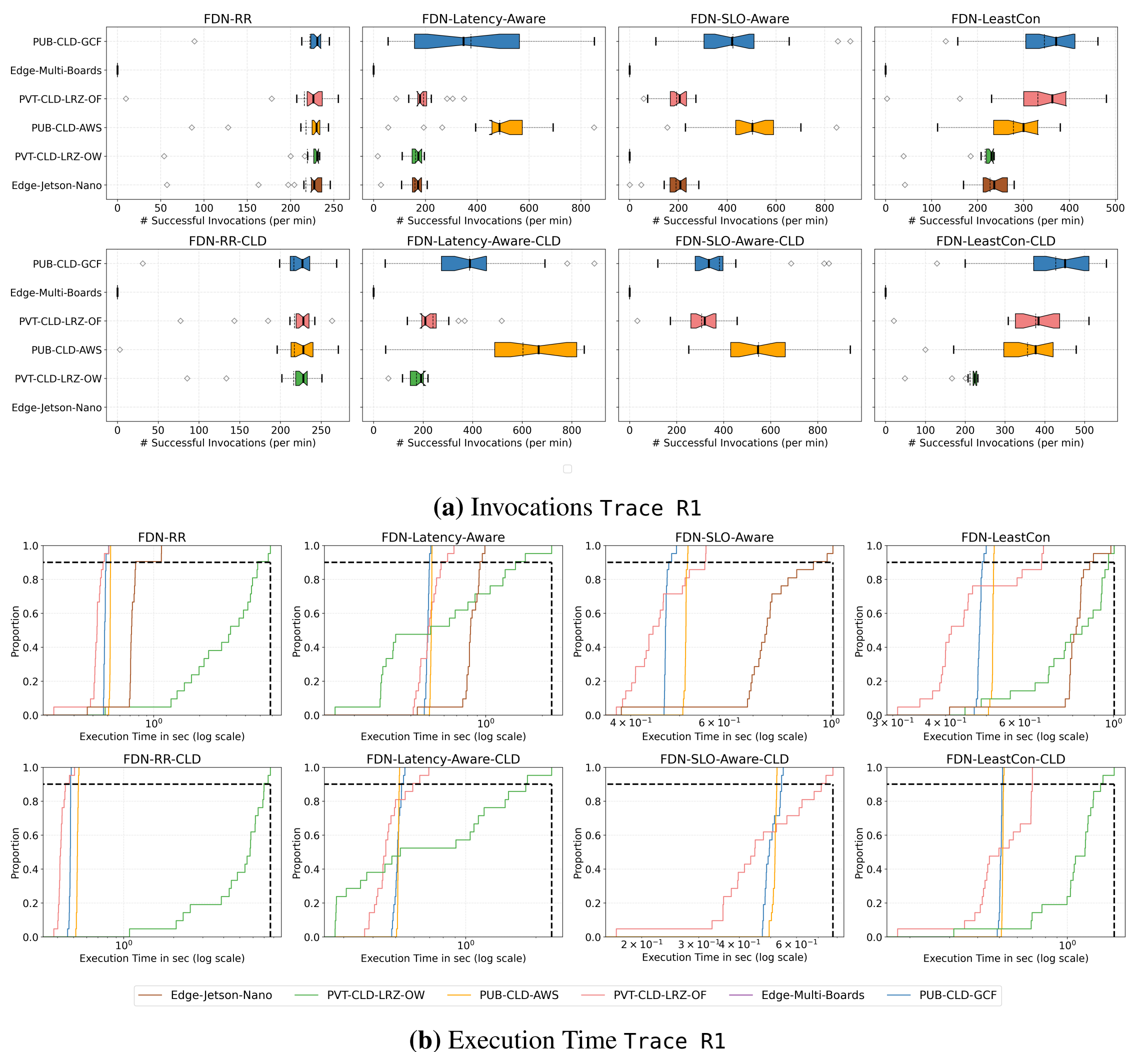
Figure 10.18.: Weights distribution among different clusters during evaluation test for `gzip-compression` function when load balanced using different algorithms for two *Invocation Traces*.

(see Figure 10.16).

FDN-Latency-Aware-Cld: We can observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `113.0` MSI/min. It is closely followed by the *PUB-CLD-GCF* cluster, which handled `112.90` MSI/min. The *PVT-CLD-LRZ-OW* cluster handled `74.23` MSI/min, while *PVT-CLD-LRZ-OF* handled `44.10` MSI/min. Furthermore, Figure 10.18b shows the weights' distribution among different clusters for `Trace R2`. The weights throughout the evaluation period are equally assigned to the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters, followed by the *PVT-CLD-LRZ-OW* cluster. We can observe a lower overall P90 response time of `0.76s` (see Figure 10.16).

FDN-SLO-Aware-Cld: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `163.04` MSI/min. It is followed by *PUB-CLD-GCF* cluster, which handled `114.38` MSI/min. Among private cloud clusters, *PVT-CLD-LRZ-OF* handled approximately `52.75` MSI/min, while *PVT-CLD-LRZ-OW* did not handle any invocation. It can again be attributed to the high cold start time in OpenWhisk. This can also be evident from Figure 10.18d, where we see the weights' distribution among different clusters for `Trace R2`. The highest weights throughout the evaluation period are assigned to the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters. For this algorithm, we can observe the overall P90 response time of `0.72`.

FDN-LeastCon-Cld: All clusters handled approximately `95` MSI/min. The overall P90 response time using this algorithm is `0.78s` (see Figure 10.16)



(a) Invocations `Trace R1`

(b) Execution Time `Trace R1`

Figure 10.19.: Details on how the successful invocations made using `Trace R1` to `gzip-compression` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

10.6.2.2 Performance on High Workload (Trace R1)

From Figure 10.16 for `Trace R1`, we observe that across the two metrics, FDN-RR has the highest P90 response time of `3.01s`. It is followed by FDN-Latency-Aware with P90 response time of `0.79s`, then FDN-LeastCon with P90 response time of`0.91s`. FDN-SLO-Aware has the lowest P90 response time of `0.72s`. Among the cloud-only algorithms, we observe that across the two metrics, again FDN-RR-Cld has the highest P90 response time of `5.51s`. It is followed by FDN-LeastCon-Cld with P90 response time of `1.04s`, and then FDN-Latency-Aware-Cld with P90 response time of `0.68s`. Again, FDN-SLO-Aware-Cld has the lowest P90 response time of `0.67s`. Across all the algorithms, again, FDN-SLO-Aware and its cloud version performed the best for `Trace R1`.

Figure 10.19a shows details on how the successful invocations made using `Trace R1` to `gzip-compression` function are distributed across each cluster, along with their execution times using different load balancing algorithms. The performance of the load balancing algorithms is presented below:

FDN-RR: Each cluster has almost handled 220 MSI/min. However, the response time on each cluster differs to a great extent. MET of invocations for the *Edge-Jetson-Nano* cluster is 0.74s, 3.16s for the *PVT-CLD-LRZ-OW*, 0.42s for the *PVT-CLD-LRZ-OF*, 0.51s for the *PUB-CLD-AWS*, and 0.47s for the *PUB-CLD-GCF*). The slowest cluster, in this case, is the *PVT-CLD-LRZ-OW* cluster, and it has impacted the performance of collaborative execution among the clusters, and that is why in Figure 10.16, we see the P90 response time for Trace R1 as 3.01s.

FDN-Latency-Aware: We can observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 487.09 MSI/min with MET of 0.51s. It is followed by the *PUB-CLD-GCF* cluster, which handled 375.57 MSI/min with MET of 0.491s. Among private cloud clusters, *PVT-CLD-LRZ-OF* handled 192.98 MSI/min with MET of 0.505s, while *PVT-CLD-LRZ-OW* handled 162.15 MSI/min with MET of 0.73s. Edge cluster *Edge-Jetson-Nano* was also not far behind, it handled 159.97 MSI/min with MET of 0.85s. Furthermore, Figure 10.18a shows the weights' distribution among different clusters (in stacked format) during the evaluation test. For Trace R1, we can observe that the highest weights during the evaluation phase are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster. The weights of the clusters varied over the evaluation phase, depending upon the execution time of invocations on the clusters. The overall P90 response time resulted by using this algorithm is 0.79s for Trace R1 (see Figure 10.16).

FDN-SLO-Aware: In this algorithm, we have set the SLO as 1s. Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 503.4 MSI/min with MET of 0.51s. It is followed by the *PUB-CLD-GCF* cluster, which handled 423.66 MSI/min with MET of 0.47s. The *PVT-CLD-LRZ-OF* cluster handled 192.72 MSI/min with MET of 0.45s. Edge cluster *Edge-Jetson-Nano* also handled around 190.7 MSI/min with MET of 0.73s. From figure 10.18c, we can observe the weights' distribution among different clusters (in stacked format). The highest weights are assigned to the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters over the entire test duration. Overall P90 response time with this algorithm is 0.72s.

FDN-LeastCon: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled 345.42 MSI/min with MET of 0.48s. It is closely followed by the *PVT-CLD-LRZ-OF* cluster which handled 331.32 MSI/min with MET of 0.454s. Then comes the *PUB-CLD-AWS* cluster which handled 278.19 MSI/min with MET of 0.51s and the *PVT-CLD-LRZ-OW* cluster handled 216.62 MSI/min with MET of 0.79s. Edge cluster *Edge-Jetson-Nano* handled more than the *PVT-CLD-LRZ-OW* cluster, around 227.14 MSI/min with MET of 0.807s. FDN-LeastCon algorithm resulted in an overall P90 response time of 0.91s (see Figure 10.16).

FDN-RR-Cld: Each cloud cluster has successfully handled the same number of invocations, approximately 217 MSI/min. The *PVT-CLD-LRZ-OF* cluster has the lowest MET of 0.42s. The slowest cluster, in this case, is the *PVT-CLD-LRZ-OW* (MET of 5.16s), which impacted the performance of collaborative execution among the clusters. The P90 response time for Trace R2 for FDN-RR-Cld is 5.51s (see Figure 10.16).

FDN-Latency-Aware-Cld: We can observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 602.8 MSI/min with MET of 0.51s. It is followed by the *PUB-CLD-GCF* cluster, which handled 387.3 MSI/min with MET of 0.511s). The private cloud cluster *PVT-CLD-LRZ-OF* handled around 239.71, with the MET of 0.47s, and the *PVT-CLD-LRZ-OW* cluster handled around 173.27, with the MET of 0.86s. Furthermore, Figure 10.18b shows the weights' distribution among different clusters (in stacked format) during evaluation test. We can observe for Trace R1, the highest weights throughout the evaluation phase are assigned to the *PUB-CLD-AWS* cluster followed by the *PUB-CLD-GCF* cluster. For this algorithm, we can observe a lower overall P90 response time of 0.68s (see Figure 10.16).

FDN-SLO-Aware-Cld: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled 548.33 MSI/min. It is followed by the *PUB-CLD-GCF* cluster, which handled 381.90 MSI/min. Among

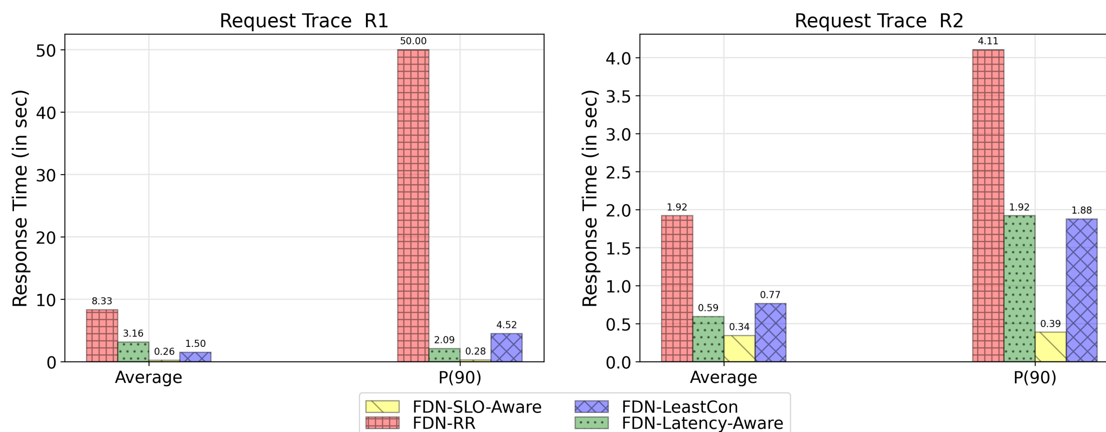


Figure 10.20.: The average and 90th percentile response times of the invocations load balanced using eight different algorithms to the `lr-prediction` function. The results are shown for both *Invocation Traces*.

private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled approximately `306.28` MSI/min, while the *PVT-CLD-LRZ-OW* cluster did not handle any invocation. It can be attributed again to the high cold start time in OpenWhisk. This can also be evident from Figure 10.18d, where we see the weights' distribution among different clusters for `Trace R1`. The highest weights throughout the evaluation period are assigned to the *PUB-CLD-AWS* cluster, followed by the *PUB-CLD-GCF* cluster. We can observe the overall P90 response time of `0.67s`.

FDN-LeastCon-Cld: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `427.52` MSI/min with MET of `0.50s`. It is closely followed by the *PVT-CLD-LRZ-OF* cluster, which handled `377.14` MSI/min with MET of `0.53s`). Then comes the *PUB-CLD-AWS* cluster, which handled `356.71` MSI/min with MET of `0.51s` and lastly the *PVT-CLD-LRZ-OW* cluster, which handled `212.36` MSI/min with MET of `1.1s`. FDN-LeastCon-Cld algorithm resulted in an overall P90 response time of `1.04s` (see Figure 10.16)

10.6.3 Individual FaaS Function (lr-prediction)

This function is deployed only on the cloud clusters (§9.2) because of its high resource requirements. Therefore, we here only show the results of four algorithms (*FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon). Figure 10.20 shows the summarized results of the evaluation, where we show the average and 90th percentile of the response times of the invocation requests, load balanced using four different algorithms.

10.6.3.1 Performance on Low Workload (Trace R2)

From Figure 10.20 for `Trace R2`, we observe that FDN-RR has the highest P90 response time of `4.11s`. It is followed by FDN-Latency-Aware with P90 response time of `1.92s`, then FDN-LeastCon with P90 response time of `1.88s`. FDN-SLO-Aware has the lowest P90 response time of `0.39s`, adhering to the defined SLO of one second.

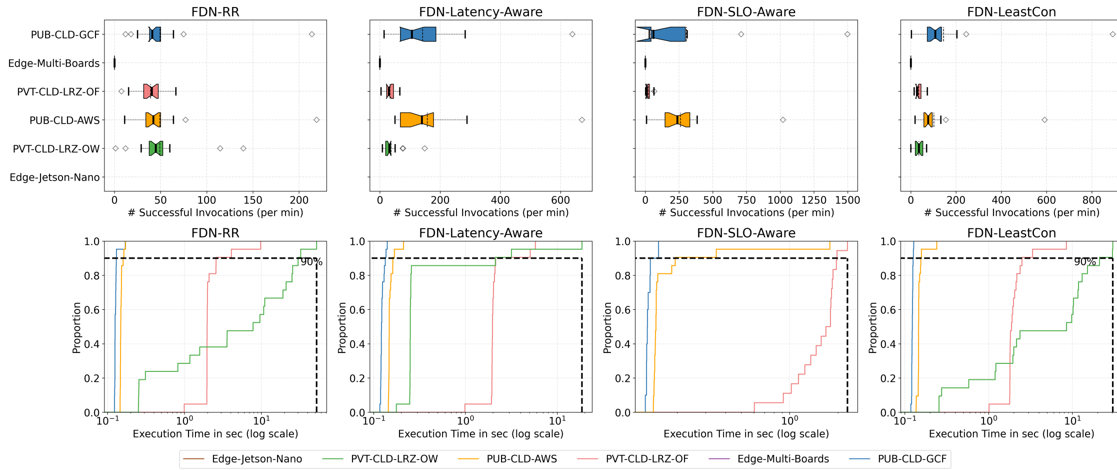
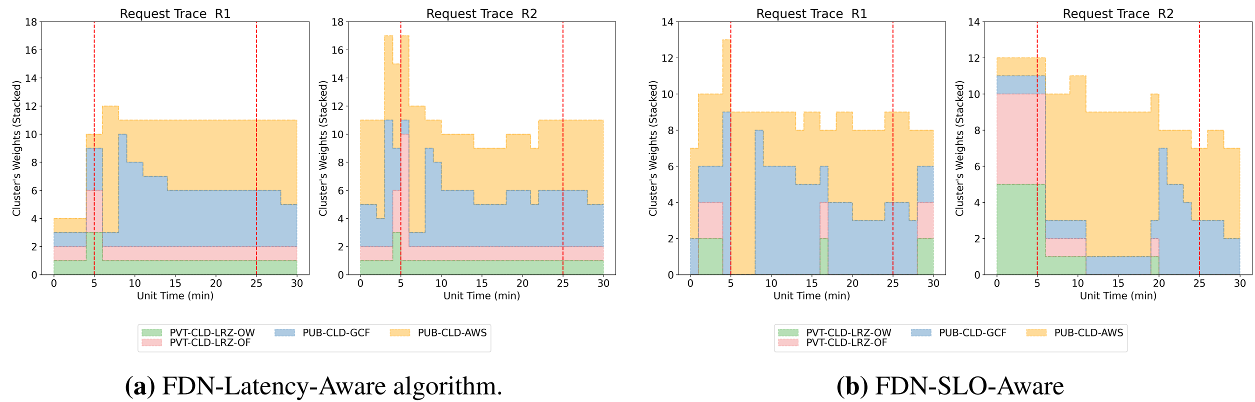


Figure 10.21.: Details on how the successful invocations made using `Trace R2` to `lr-prediction` function are distributed across each cluster, along with their execution times using different load balancing algorithms.



(a) FDN-Latency-Aware algorithm.

(b) FDN-SLO-Aware

Figure 10.22.: Weights distribution among different clusters during evaluation test for `lr-prediction` function when load balanced using different algorithms for two *Invocation Traces*.

Figure 10.21 shows details on how the successful invocations made using `Trace R2` to `lr-prediction` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

Performance of the load balancing algorithms is presented below:

FDN-RR: Each cluster has successfully handled almost `50` MSI/min, however with different execution times. The overall P90 response time for `Trace R2` for FDN-RR is `1.92s` (see Figure 10.20). The *PUB-CLD-GCF* cluster has the lowest MET of `0.12s`, followed by the *PUB-CLD-AWS* cluster with MET of `0.15s`, then comes the two private cloud clusters. The MET of invocations on the *PVT-CLD-LRZ-OW* cluster is `12.22s`, while on the *PVT-CLD-LRZ-OF* cluster is `2.46s`. The overall P90 response time following this algorithm is `4.11s`.

FDN-Latency-Aware: In case of FDN-Latency-Aware algorithm, we can observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `157.14` MSI/min with MET of `0.15s`. It is closely followed by the *PUB-CLD-GCF* cluster, which handled `141.7` MSI/min with MET of `0.12s`. Both private cloud clusters, handled approximately `35` MSI/min. The *PVT-CLD-LRZ-OF* cluster handled with

MET of `2.3s`, while the *PVT-CLD-LRZ-OW* cluster handled with MET of `1.35s`. The weight assignment across clusters during the evaluation can be observed in Figure 10.22a. We can observe for `Trace R2` that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster followed by the *PUB-CLD-GCF* cluster, which also verifies with the number of invocations handled by each cluster. The other two cloud clusters are assigned one weight throughout the evaluation period. Overall P90 response time by using this algorithm is `1.92s` (see Figure 10.20).

FDN-SLO-Aware: Here, most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `304.3` MSI/min with MET of `0.13s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `261.0` MSI/min with MET of `0.23s`. Among private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled approximately `21.55` MSI/min with MET of `1.56s`, while the *PVT-CLD-LRZ-OW* cluster did not handle any invocation. It can be attributed to the high cold start time in OpenWhisk. The invocations distributions are based on the weights assigned to the cluster, and Figure 10.22b shows the weights' distribution among different clusters (in stacked format). We can observe that for `Trace R2`, the highest weights throughout the evaluation are assigned to the *PUB-CLD-GCF* and *PUB-CLD-AWS* clusters. During the initial period, we observe that the *PVT-CLD-LRZ-OW* cluster is assigned one weight. The invocations resulting from that assigned weight resulted in errors. FDN-SLO-Aware algorithm for `Trace R2` resulted in an overall P90 response time of `0.39s` (see Figure 10.20), adhering to the defined SLO of one second.

FDN-LeastCon: Most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `144.761` MSI/min with MET of `0.123s`. It is closely followed by the *PUB-CLD-AWS* cluster, which handled `102.28` MSI/min with MET of `0.151s`. Both private cloud clusters, handled approximately `35` MSI/min. The *PVT-CLD-LRZ-OF* cluster handled with MET of `2.2s`, while the *PVT-CLD-LRZ-OW* cluster handled with MET of `8.78s`. FDN-LeastCon algorithm for `Trace R2` resulted in an overall P90 response time of `1.88s` (see Figure 10.20).

10.6.3.2 Performance on High Workload (Trace R1)

From Figure 10.20 for `Trace R1`, we observe that the FDN-RR has the highest P90 response time of `50.0s`, reaching the maximum set value. It is followed by FDN-LeastCon with P90 response time of `4.52s`, then comes the FDN-Latency-Aware with P90 response time of `2.09s`. FDN-SLO-Aware has the lowest P90 response time of `0.28s`, adhering to the defined SLO of one second.

Figure 10.23 shows details on how the successful invocations made using `Trace R1` to the `lr-prediction` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

FDN-RR: Each cluster has almost successfully handled the same number of invocations, except *PVT-CLD-LRZ-OW*. The *PVT-CLD-LRZ-OW* cluster handled `119.66` MSI/min with MET of `20.9s`. It handled doubled the amount of invocations as compared to other clusters. We do not have concrete reasoning for this behavior, but we assume that the other clusters returned errors for the rest of the invocations. The *PVT-CLD-LRZ-OF* cluster handled `55.5` MSI/min with MET of `2.59s`, the *PUB-CLD-AWS* cluster handled `60.52` MSI/min with MET of `0.306s`, and the *PUB-CLD-GCF* cluster handled `60.09` MSI/min with MET of `0.128s`. The slowest cluster in this case is the *PVT-CLD-LRZ-OW* and that's why in Figure 10.20, we see a high P90 response time for `Trace R1` for FDN-RR as `50s`.

FDN-Latency-Aware: We can observe that most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `234.6` MSI/min with MET of `0.163s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `195.9` MSI/min with MET of `0.12s`. Then comes the two private clusters, the *PVT-CLD-LRZ-OW* cluster handled `83.16` MSI/min with MET of `20.38`, while the *PVT-CLD-LRZ-OF* cluster handled

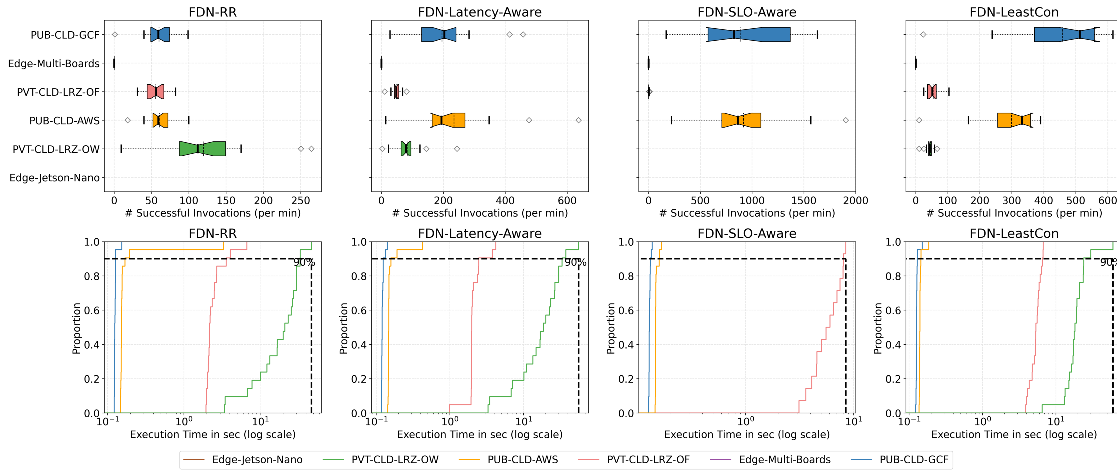


Figure 10.23.: Details on how the successful invocations made using `Trace R1` to `lr-prediction` function are distributed across each cluster, along with their execution times using different load balancing algorithms.

`48.75` MSI/min with MET of `2.19s`. The invocations distributions are based on the execution duration of the function, and it can further be evident from Figure 10.22a, where we show the weights' distribution among different clusters (in stacked format). We can observe that for `Trace R1`, the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters. The other two cloud clusters are assigned a weight of one throughout the evaluation period. FDN-Latency-Aware algorithm resulted in an overall P90 response time of `2.09s` (see Figure 10.20).

FDN-SLO-Aware: Most of the invocations are handled by the *PUB-CLD-AWS* cluster, which handled `916.90` MSI/min with MET of `0.14s`. It is closely followed by the *PUB-CLD-GCF* cluster, which handled `885.76` MSI/min with MET of `0.12s`. Among private cloud clusters, the *PVT-CLD-LRZ-OF* cluster handled approximately `3` MSI/min, while *PVT-CLD-LRZ-OW* did not handle any invocation. It can be attributed again to the high cold start time in OpenWhisk. The invocations distributions are based on the weights assigned to the cluster, and Figure 10.22b shows the weights' distribution among different clusters (in stacked format). We can observe that for `Trace R1`, the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters. FDN-SLO-Aware algorithm for `Trace R1` resulted in an overall P90 response time of `0.28s` (see Figure 10.20), adhering to the defined SLO of one second.

FDN-LeastCon: In this algorithm, most of the invocations are handled by the *PUB-CLD-GCF* cluster, which handled `459.0` MSI/min with MET of `0.12s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `299.14` MSI/min with MET of `0.14s`. Then comes the *PVT-CLD-LRZ-OF* cluster which handled only `54.17` MSI/min with MET of `5.41`, followed by the *PVT-CLD-LRZ-OW* cluster which handled `43.5` MSI/min with MET of `20.46`. FDN-LeastCon algorithm resulted in an overall P90 response time of `4.52s` (see Figure 10.20).

10.6.4 Serverless Application (faas-composer)

All the eight functions within the serverless application (`faas-composer`) are deployed solely on the cloud clusters (§9.2). Therefore, we show the results of four algorithms (*FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon). When evaluating an algorithm for `faas-composer`, all the functions it invokes are load balanced using the same algorithm. Since `faas-composer` needs to send requests to FDN

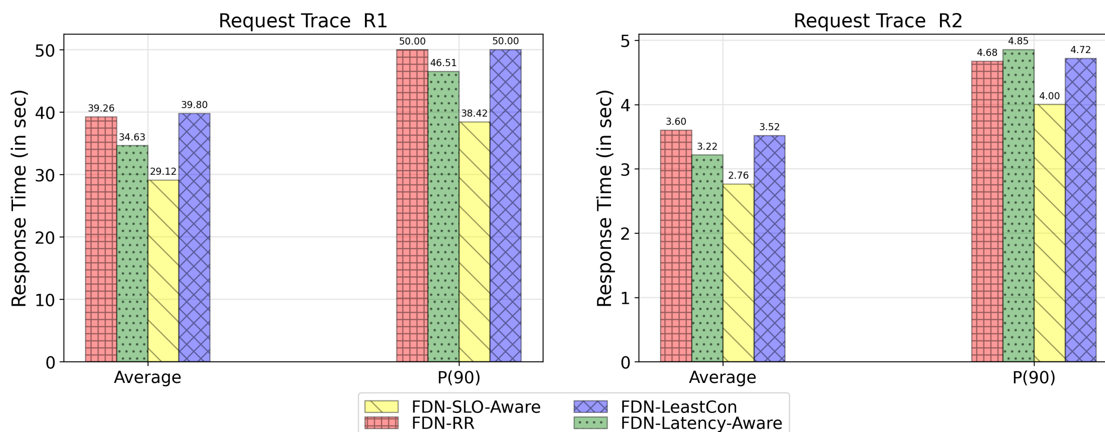


Figure 10.24.: The average and 90th percentile of the response times of the invocation requests, load balanced using four different algorithms to the `faas-composer` function. The request traces are lowered-down versions of the original ones, with the maximum number of requests per second as 10.

endpoint for the functions invocations and *PVT-CLD-LRZ-OW* cluster is in the same network as the FDN, therefore, it was deployed only in the *PVT-CLD-LRZ-OW* cluster. In our evaluation of `faas-composer`, we initially used both original *Invocation Traces*, but many invocations timed out. We suspect that each private cloud cluster is limited in the number of resources, and when all the functions scale, they contend with each other for resources. As a result, the execution time increases, and invocations time out. It could also be possible that the `faas-composer` is unsuitable for the OpenWhisk platform. We did not investigate it further, since the *PVT-CLD-LRZ-OW* cluster was the only cluster that could send invocations to the FDN endpoint. Furthermore, a function requiring high compute power could become the bottleneck in the `faas-composer` and influence the time-outs. Nevertheless, for evaluating `faas-composer`, we lowered-down the maximum number of invocations per second to `10` in both `R1` and `R2` traces.

Figure 10.24 shows the average and 90th percentile response times of the invocations, load balanced using four different algorithms. For `R1`, we observe that FDN-RR and FDN-LeastCon has the highest P90 response time of `50.0s`, reaching the maximum set value. It is followed by FDN-Latency-Aware with P90 response time of `46.51s`. FDN-SLO-Aware has the lowest P90 response time of `38.42s`. All the algorithms have a high P90 response time. For the `Trace R2`, we observe that FDN-Latency-Aware has the highest P90 response time of `4.85s`. It is followed by FDN-LeastCon with P90 response time of `4.72s`, then comes the FDN-RR with P90 response time of `4.68s`. FDN-SLO-Aware again, has the lowest P90 response time of `4s`.

10.6.4.1 Performance on Low Workload (lowered-down version of Trace R1)

Figure 10.25 shows details on how the successful invocations made using the lowered-down version of `Trace R1` to the `faas-composer` are distributed across each function (eight columns showing eight functions) and clusters using different load balancing algorithms (rows showing different algorithms, from top to bottom: *FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon). Furthermore, Figure 10.26 shows the corresponding execution times. We now analyze the performance of each algorithm:

FDN-RR: In Figure 10.25, from the first row we observe that, for each function, each cluster has almost successfully handled `6` MSI/min. The performance of the clusters for each function is described below:

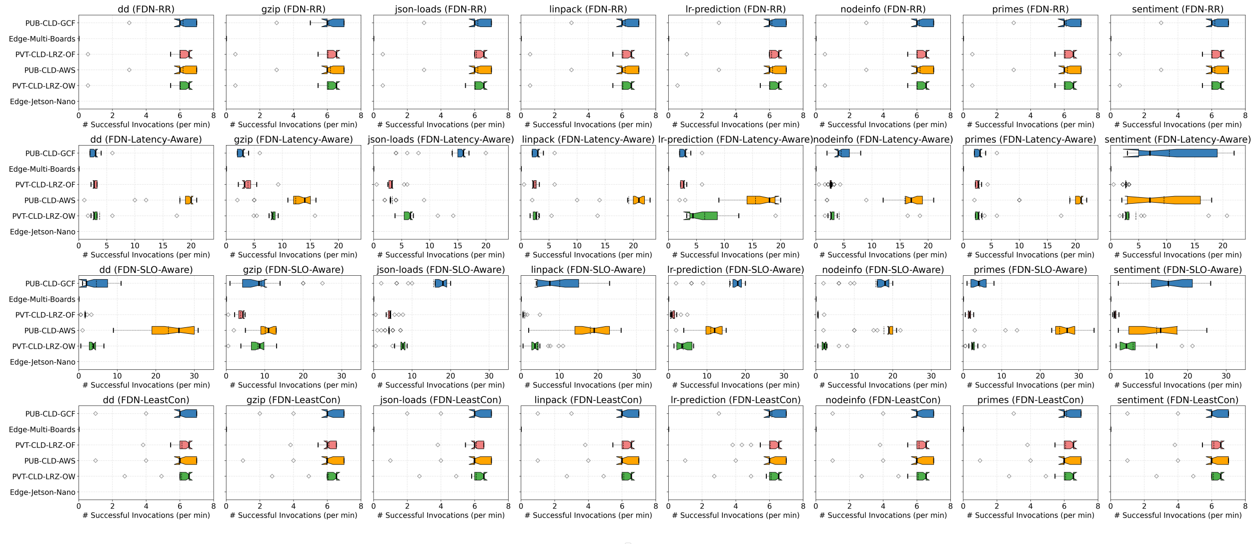


Figure 10.25.: Details on how the successful invocations load tested using lowered-down version of `Trace R1` to `faas-composer` following different algorithms are distributed across the functions and clusters. The rows show the different algorithms (From top to bottom: *FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon) and columns show different functions.

- `dd`: Each cluster has handled `6` MSI/min. The *PUB-CLD-AWS* cluster handled with the best MET of `0.01s`, followed by *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.024s`, then comes the *PUB-CLD-GCF* cluster, which handled with the MET of `0.11s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the worst MET of `0.42s`.
- `gzip-compression`: Again, each cluster has handled `6` MSI/min. The *PVT-CLD-LRZ-OW* cluster handled with the best MET of `0.28s`, followed by *PUB-CLD-GCF* cluster, which handled with the MET of `0.49s`, then comes the *PUB-CLD-AWS* cluster, which handled with the MET of `0.51s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the worst MET of `0.66s`.
- `json-loads`: Each cluster has handled `6` MSI/min. The *PUB-CLD-GCF* cluster handled with the best MET of `0.18s`, followed by *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.26s`, then comes the *PVT-CLD-LRZ-OF* cluster, which handled with the MET of `0.60s`,and lastly *PUB-CLD-AWS* cluster handled with the worst MET of `1.0s`.
- `linpack`: The *PUB-CLD-AWS* cluster handled with the best MET of `0.003s`, followed by *PUB-CLD-GCF* cluster, which handled with the MET of `0.0049s`, then comes the *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.0078s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the worst MET of `0.32s`.
- `lr-prediction`: The *PUB-CLD-GCF* cluster handled with the best MET of `0.12s`, followed by *PUB-CLD-AWS* cluster, which handled with the MET of `0.16s`, then comes the *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.24s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the worst MET of `1.75s`.
- `nodeinfo`: The *PUB-CLD-AWS* cluster handled with the best MET of `0.0015s`, followed by *PUB-CLD-GCF* cluster, which handled with the MET of `0.0034s`, then comes the *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.005s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the MET of `0.15s`.

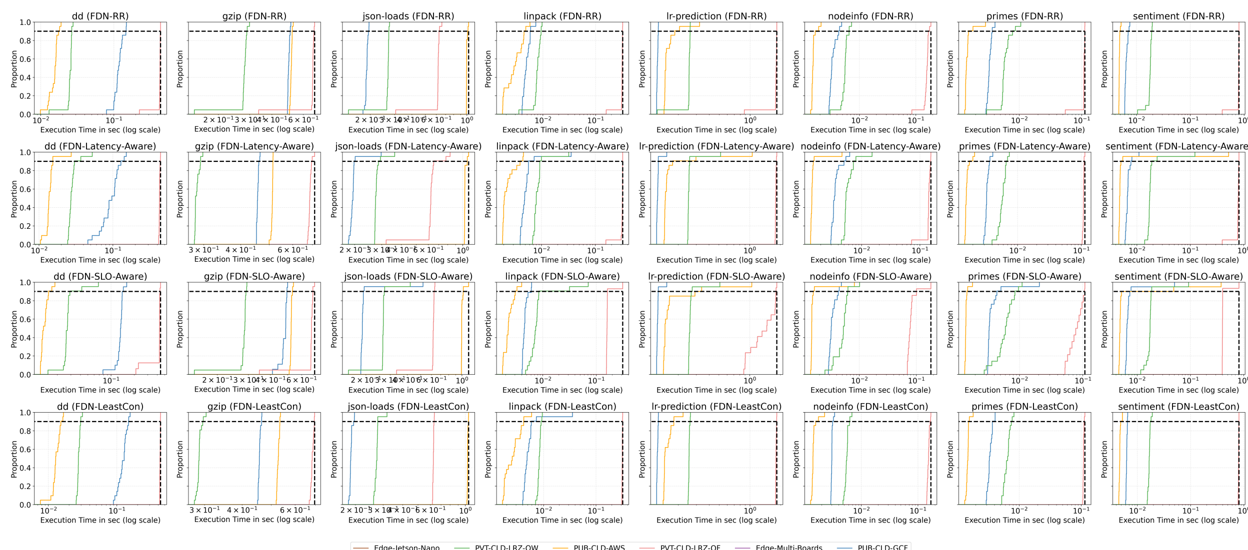


Figure 10.26.: Details on how the successful invocations' execution time load tested using lowered-down version of `Trace R1` to `faas-composer` application following different algorithms are distributed across the functions and clusters. The rows show the different algorithms (From top to bottom: *FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon) and columns show different functions.

- `primes`: The *PUB-CLD-AWS* cluster handled with the best MET of `0.001s`, followed by *PUB-CLD-GCF* cluster, which handled with the MET of `0.0015s`, then comes the *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.0061s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the MET of `0.107s`.
- `sentiment-analysis`: The *PUB-CLD-AWS* cluster handled with the best MET of `0.0048s`, followed by *PUB-CLD-GCF* cluster, which handled with the MET of `0.0065s`, then comes the *PVT-CLD-LRZ-OW* cluster, which handled with the MET of `0.0177s`,and lastly *PVT-CLD-LRZ-OF* cluster handled with the MET of `0.776s`.

In summary, for FDN-RR, we observe that for each function, each cluster has successfully handled `6` MSI/min. Among all the clusters, the *PUB-CLD-AWS* cluster has performed the best for most of the functions. The *PVT-CLD-LRZ-OF* cluster has the worst performance among all the clusters. Even though for each function the MET was below one second, from Figure 10.24, we see a high overall P90 response time for `Trace R1` as `50s`. When combining all the functions' responses, it could also be possible that the `faas-composer` takes a longer time. It is also possible that `faas-composer` is unsuitable for the OpenWhisk platform. Additionally, we measure the performance of each function in terms of MET; it could be possible that the response time to `faas-composer` has a higher latency. To investigate this, we need to add profiling in `faas-composer`, which is out of the scope of current work.

FDN-Latency-Aware: In Figure 10.25, the second row represents the invocations handled by each cluster for every function when load balanced using this algorithm. We observe that, across all the functions, the *PUB-CLD-AWS* cluster handled most of the invocations. The invocations distributions are based on the weights assigned to the cluster. The first row in Figure 10.27 shows the weights' distribution among different clusters (in stacked format) for all the functions. Next, we present a detailed analysis of the performance of each function.

- `dd`: The *PUB-CLD-AWS* cluster handled the most number of invocations (`18` MSI/min) with MET of `0.014s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `3.71` MSI/min with MET

of `0.028s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-GCF* clusters, both handled approximately `2.8` MSI/min with MET of `0.428s` and `0.096s` respectively. These invocations distributions can also be evident from the first row in Figure 10.27, where the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster, while all the other clusters have a weight of one.

- `gzip-compression`: For this function, again the *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `12.3` MSI/min with MET of `0.51s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `8.44` MSI/min with MET of `0.284s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-GCF* clusters, both handled approximately `3.5` MSI/min with MET of `0.68s` and `0.45s` respectively. From the first row in Figure 10.27, we observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster. It is followed by the *PVT-CLD-LRZ-OW* cluster, while the other two clusters have a weight of one throughout the evaluation.
- `json-loads`: The *PUB-CLD-GCF* cluster handled the most number of invocations, which handled `14.04` MSI/min with MET of `0.20s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `6.57` MSI/min with MET of `0.28s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-AWS* clusters, both handled approximately `3.2` MSI/min with MET of `0.62s` and `1.03s` respectively. From the first row in Figure 10.27, we can observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-GCF* cluster. It is followed by the *PVT-CLD-LRZ-OW* cluster, while the other two clusters have a weight of one throughout the evaluation.
- `linpack`: For this function, again the *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `19.2` MSI/min with MET of `0.002s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `3.142` MSI/min with MET of `0.009s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-GCF* clusters, both handled approximately `2.6` MSI/min with MET of `0.310s` and `0.0064s` respectively. These invocations distributions can also be evident from the first row in Figure 10.27, where the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster, while all the other clusters have a weight of one.
- `lr-prediction`: The *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `15.5` MSI/min with MET of `0.2s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `6.46` MSI/min with MET of `0.26s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-GCF* clusters, both handled approximately `2.8` MSI/min with MET of `1.802s` and `0.14s` respectively. Even though the MET on the *PUB-CLD-GCF* cluster is lowest, it still did not handle the highest number of invocations. These invocations distributions can also be evident from the first row in Figure 10.27, where the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster. It is followed by the *PVT-CLD-LRZ-OW* cluster, while the other two clusters have a weight of one throughout the evaluation.
- `nodeinfo`: The *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `15.8` MSI/min with MET of `0.001s`. It is followed by the *PVT-CLD-LRZ-OW* and *PUB-CLD-GCF* clusters, both handled approximately `4.5` MSI/min with MET of `0.0062s` and `0.0036s` respectively. Then comes the *PVT-CLD-LRZ-OF* cluster, which handled `2.64` MSI/min with MET of `0.140s`. From the first row in Figure 10.27, we can observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster. It is followed by the *PVT-CLD-LRZ-OW* and *PUB-CLD-GCF* clusters, while the *PVT-CLD-LRZ-OF* clusters have a weight of one throughout the evaluation throughout the evaluation.
- `primes`: The *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `18.9` MSI/min with MET of `0.001s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `3.5`

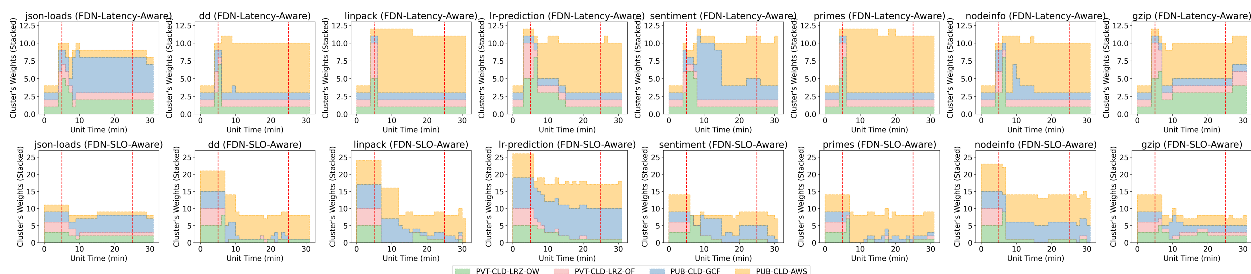


Figure 10.27.: Weights distribution among different clusters during evaluation test for `faas-composer` function when load balanced using different algorithms for two *Invocation Traces*.

MSI/min with MET of `0.0058s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-GCF* clusters, both handled approximately `2.5` MSI/min with MET of `0.108s` and `0.0033s` respectively. From the first row in Figure 10.27, we observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster, and is followed by all the other clusters having a weight of one throughout the evaluation. Even though the *PUB-CLD-GCF* has a lower MET than the *PVT-CLD-LRZ-OF* cluster, but it still handled lower number of invocations. This is due to `faas-composer` being running on the same cluster as *PVT-CLD-LRZ-OF*, as a result it has a lower latency and is able to process more requests.

- `sentiment-analysis`: The *PUB-CLD-GCF* cluster handled the most number of invocations, which handled `10.47` MSI/min with MET of `0.0068s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `9.52` MSI/min with MET of `0.0295s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `4.51` MSI/min with MET of `0.022s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled `2.658` MSI/min with MET of `0.77s`. These invocations distributions can also be evident from the first row in Figure 10.27, where the highest weights throughout the evaluation are assigned to the *PUB-CLD-GCF* and *PUB-CLD-AWS* clusters. It is followed by the *PVT-CLD-LRZ-OW* cluster, while the *PVT-CLD-LRZ-OF* cluster has a weight of one throughout the evaluation.

In summary, for FDN-Latency-Aware, we observe that across all functions, the *PUB-CLD-AWS* cluster has handled the most number of invocations except for `sentiment-analysis` and `json-loads` functions, where the *PUB-CLD-GCF* cluster handled the most number of invocations. This can also be easily seen by looking at the first row in Figure 10.27, where we show the weights assigned to the cluster, and the *PUB-CLD-AWS* cluster dominates across most of the functions. The *PVT-CLD-LRZ-OF* cluster has the worst performance among all the clusters. Again, even though for each function the MET was below one second, from Figure 10.24, we see a high overall P90 response time for `Trace R1` as `46.51s`. We suspect that here measure the performance of each function in terms of MET, while in Figure 10.24, we look at the P90 response time of the `faas-composer` function. It could be possible that the response times from individual functions to `faas-composer` have higher latencies. Nevertheless, this investigation is out of the scope of current work.

FDN-SLO-Aware: Figure 10.25, the third row represents the invocations handled by each cluster for every function when load balanced using this algorithm. We observe that, across all the functions, the *PUB-CLD-AWS* cluster handled most of the invocations. The invocations distributions are based on the weights assigned to the cluster. The second row in Figure 10.27 shows the weights' distribution among different clusters (in stacked format) for all the functions. Next, we present a detailed analysis of the performance of each function.

- `dd`: The *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `23.28` MSI/min with MET of `0.013s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `4.6`

MSI/min with MET of `0.13s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `3.52` MSI/min with MET of `0.028s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled `1.636` MSI/min with MET of `0.40s`. From the second row in Figure 10.27, we can observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster. In the initial period of evaluation, high weights were assigned to *PUB-CLD-GCF* as well, but over the course, it got reduced to zero with occasional spikes. *PVT-CLD-LRZ-OW* cluster has one weight for the most period of evaluation.

- `gzip-compression`: For this function, the *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `10.14` MSI/min with MET of `0.52s`. It is closely followed by the *PUB-CLD-GCF* cluster, which handled `9.08` MSI/min with MET of `0.47s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `8.28` MSI/min with MET of `0.27s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled `3.75` MSI/min with MET of `0.67s`. From the second row in Figure 10.27, we observe that throughout the evaluation, the weights are equally shared among three clusters (*PUB-CLD-AWS*, *PUB-CLD-GCF* and *PVT-CLD-LRZ-OW*), while one weight is assigned to the *PUB-CLD-AWS* cluster for the entire evaluation.

- `json-loads`: The *PUB-CLD-GCF* cluster handled the most number of invocations, which handled `15.6` MSI/min with MET of `0.20s`. It is followed by the *PVT-CLD-LRZ-OW* cluster, which handled `7.14` MSI/min with MET of `0.26s`. Then comes the *PVT-CLD-LRZ-OF* and *PUB-CLD-AWS* clusters, both handled `4` MSI/min with MET of `0.60s` and `1.03s` respectively. From the second row in Figure 10.27, we can observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-GCF* cluster. It is followed by the *PVT-CLD-LRZ-OW* cluster, while the other two clusters have a weight of one throughout the evaluation.

- `linpack`: For this function, the *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `18.0` MSI/min with MET of `0.0025s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `10.05` MSI/min with MET of `0.004s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `4.25` MSI/min with MET of `0.011s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled only `1` MSI/min with MET of `0.17s`. These invocations distributions can also be evident from the second row in Figure 10.27, where most of the weights are shared between *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters, with *PUB-CLD-AWS* having the highest weights. The weights for the *PVT-CLD-LRZ-OW* cluster are increased around the 18-minute mark in the figure.

- `lr-prediction`: The *PUB-CLD-GCF* cluster handled the most number of invocations, which handled `15.9` MSI/min with MET of `0.12s`. It is closely followed by the *PUB-CLD-AWS* cluster, which handled `11.00` MSI/min with MET of `0.21s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `3.86` MSI/min with MET of `0.27s`. These invocations distributions can also be evident from the second row in Figure 10.27, where the highest weights throughout the evaluation are assigned to the *PUB-CLD-GCF* cluster. It is followed by the *PUB-CLD-AWS* and *PVT-CLD-LRZ-OW* clusters.

- `nodeinfo`: The *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `17.7` MSI/min with MET of `0.001s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `15.61` MSI/min with MET of `0.003s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `2.40` MSI/min with MET of `0.0052s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled only `1` MSI/min with MET of `0.0855s`. From the second row in Figure 10.27, we can observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster. It is followed by the *PUB-CLD-GCF* cluster and *PVT-CLD-LRZ-OW* cluster, while the *PVT-CLD-LRZ-OF* clusters has a weight of one occasionally during the evaluation.

- `primes`: The *PUB-CLD-AWS* cluster handled the most number of invocations, which handled `25.04` MSI/min with MET of `0.001s`. It is followed by the *PUB-CLD-GCF* cluster, which handled `4.3`

MSI/min with MET of `0.0043s`. Then comes the *PVT-CLD-LRZ-OW* and *PVT-CLD-LRZ-OF* clusters, both handled approximately `2` MSI/min with MET of `0.006s` and `0.079s` respectively. From the second row in Figure 10.27, we observe that the highest weights throughout the evaluation are assigned to the *PUB-CLD-AWS* cluster, and is followed by *PUB-CLD-GCF* cluster, while the other clusters were assigned a weight of one occasionally during the evaluation.

- `sentiment-analysis`: The *PUB-CLD-GCF* cluster handled the most number of invocations, which handled `15.10` MSI/min with MET of `0.008s`. It is followed by the *PUB-CLD-AWS* cluster, which handled `11.95` MSI/min with MET of `0.025s`. Then comes the *PVT-CLD-LRZ-OW* cluster, which handled `5.84` MSI/min with MET of `0.020s`. Lastly, the *PVT-CLD-LRZ-OF* cluster handled `1.6` MSI/min with MET of `0.42s`. These invocations distributions can also be evident from the second row in Figure 10.27, where the highest weights throughout the evaluation are assigned to the *PUB-CLD-GCF* and *PUB-CLD-AWS* clusters. It is followed by the *PVT-CLD-LRZ-OW* cluster, while the *PVT-CLD-LRZ-OF* cluster has almost negligible weight assignment throughout the evaluation.

The invocations distributions are based on the weights assigned to the cluster. The second row in Figure 10.27 shows the weights' distribution among different clusters (in stacked format) for different functions. From Figure 10.24, we see an overall P90 response time for `Trace R1` for FDN-SLO-Aware as `38.42s`.

In summary, for FDN-SLO-Aware, we observe that across all functions, the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters handled the most number of invocations. This can also be seen in the second row of Figure 10.27, where we show the weights assigned to the cluster, and the *PUB-CLD-AWS* and *PUB-CLD-GCF* clusters dominate across most of the functions. The *PVT-CLD-LRZ-OF* cluster has the worst performance among all the clusters. Again, even though for each function the MET was below one second, from Figure 10.24, we see a high overall P90 response time for `Trace R1` as `38.42s`. We suspect that the response times from individual functions to `faas-composer` have higher latencies. Additionally, even though the SLO was assigned one second, this algorithm could not provide the response time adhering to the requirements. However, the function execution time was much higher than one second, a readjustment to the desired SLO is required, and this investigation is out of the scope of current work. When comparing FDN-SLO-Aware with FDN-Latency-Aware, we can see from Figure 10.24, that FDN-SLO-Aware performed better but overall P90 response time is still very high.

FDN-LeastCon: Figure 10.25, the fourth row represents the invocations handled by each cluster for every function when load balanced using this algorithm. We observe that this algorithm behaves similarly to the FDN-RR algorithm. For each function, each cluster has almost successfully handled `6.1` MSI/min. Since there were not many invocations, we suspect that the open connections are the same among all clusters, which is why this algorithm performs similarly to the FDN-RR algorithm. Thus, from Figure 10.24, we see an overall P90 response time for `Trace R1` for FDN-LeastCon is also `50s`, as FDN-RR.

10.7 FDN's Load Balancing Performance Summary

FDN's **Courier** component already provides benefits for the cluster administrators for collaboration between multiple heterogeneous target clusters. It helps to overcome the shortcomings of individual target clusters, such as edge clusters, by shifting the invocations to the cloud clusters (as seen for *nodeinfo* function in §10.6.1). This collaboration mechanism can also be used to reduce the cold-start problem. It can be done by keeping a low resource cluster always warm, directing initial function invocations to it, and later using weighted collaboration between other target clusters. Moreover, it is also possible to create a dynamic rule inside the load balancer that checks for the warm target platform and directs the initial function invocations to it, leading to better performance. In our evaluation (§10.6), we used multiple load balancing algorithms

Table 10.11.: Summary of the load balancing algorithms results on the FaaS functions.

Function	-	FDN-RR	FDN-Latency-Aware	FDN-SLO-Aware	FDN-Least-Con	FDN-RR-Cld	FDN-Latency-Aware-Cld	FDN-SLO-Aware-Cld	FDN-Least-Con-Cld
`nodeinfo`	R1	0.51	0.45	0.14	0.44	0.15	0.14	**0.13**	0.14
	R2	0.50	0.48	**0.14**	0.48	0.16	0.15	**0.14**	0.15
`primes`	R1	0.36	0.34	0.14	0.33	**0.13**	**0.13**	**0.13**	**0.13**
	R2	0.38	0.35	0.14	0.36	0.14	0.14	0.14	**0.13**
`linpack`	R1	10.06	1.09	0.14	0.33	0.39	0.31	**0.13**	0.15
	R2	1.5	1.06	0.15	1.04	0.35	0.25	**0.14**	0.32
`sentiment`	R1	15.98	0.84	**0.14**	**0.14**	4.31	0.24	**0.14**	4.48
	R2	2.95	0.85	**0.14**	0.81	0.84	0.26	**0.14**	0.87
`dd`	R1	7.02	1.65	**0.15**	0.44	0.45	0.41	0.16	0.40
	R2	1.63	1.46	**0.18**	1.41	0.44	0.38	0.25	0.47
`gzip`	R1	3.01	0.79	0.72	0.91	5.51	0.68	**0.67**	1.04
	R2	1.66	1.55	1.03	1.42	0.75	0.76	**0.72**	0.78
`json-loads`	R1	16.95	6.32	1.55	5.75	5.81	6.8	**1.12**	17.2
	R2	1.92	1.93	1.65	1.88	2.27	2.12	**0.91**	1.89
`lr-prediction`	R1	50	2.09	**0.28**	4.52	-	-	-	-
	R2	4.11	1.92	**0.39**	1.88	-	-	-	-
`image-process`	R1	50	**0.53**	0.65	0.65	-	-	-	-
	R2	0.69	0.65	**0.55**	0.66	-	-	-	-

for collaboration between multiple heterogeneous target clusters. In the following subsections, we discuss the performance of the load balancing algorithms in §10.7.1 and clusters performance in §10.7.2.

10.7.1 Algorithms Performance

In Table 10.11, we present the summarized results of each algorithm, showcasing the P90 response time for each function for both request traces. One can observe that the dynamic algorithms (FDN-LeastCon, FDN-SLO-Aware FDN-Latency-Aware, and their cloud-centric versions) made better load-balancing decisions than the traditional load-balancing FDN-RR algorithm. It means that using the dynamic algorithms, administrators of FDN can easily add new clusters, and the **Courier's** load-balancing algorithms would automatically adapt. In traditional systems that use FDN-RR algorithms, changing requirements might require a manual re-evaluation of the weights. However, in our case, **Courier** adapts the weights for clusters according to their performance. If the functions and requirements for developers change, the system automatically adapts to them. In general, from Table 10.11, we can infer that, FDN-SLO-Aware and its cloud variant performed the best among all the algorithms and for all functions. Furthermore, since cloud-centric algorithms have the advantage of load balancing across high compute capable cloud clusters, we see they have the lowest P90 response time as their normal counterparts. We now present a detailed analysis of each algorithm.

FDN-RR: In this case, the function invocations are distributed across all target clusters in a round-robin manner. One can observe from Table 10.11 that, in all cases, it has performed the worst due to its inability

to adapt to the runtime situation. Furthermore, with the increase in user workload invocations, its performance got worse, and the P90 response time has increased (except for *primes-python* function). The biggest drawback of using the round-robin approach is that it assumes target platforms are similar enough to handle equivalent loads. However, because of heterogeneous target platforms in the FDN, the algorithm has no way to distribute more or fewer requests to these target platforms based on their resources. As a result, target platforms with less capacity may overload and fail more quickly, while capacity on other platforms remains idle. Therefore, in this case, we require weighted collaboration, where function invocations are distributed across the target cluster based on the weights assigned to each target cluster.

FDN-Latency-Aware: In all the evaluation scenarios, we observed that the weight generation based on the average execution times of the function running in a cluster could provide good insight into the computational capabilities of the cluster. Furthermore, the automatic dynamic changes in weights for each cluster (For example, in Figure 10.18b for *gzip-comporession* function) during the testing scenarios show the algorithm's adaptability if a cluster is overloaded. From Table 10.11, we can see that the P90 response time for this algorithm is much lower than that of FDN-RR. Furthermore, with the increase in user requests (R1 request trace), this algorithm can adapt and deliver approximately the same response time as that with fewer user requests (R2 request trace). However, the frequent collection of metrics from each cluster can induce an additional overhead on the cluster, which can be avoided by setting the optimal weight update δ time. Additionally, if the differences in runtime are not big, all the clusters will be assigned with the same weights. Currently, the algorithm always assigns at least weight 1 to a cluster. This is done to allow the requests to reach all clusters and collaborate. However, if a cluster is overloaded or not working, the requests would still go to it. For example, in Figure 10.19a for *gzip-compression* function, where some requests were still going to the *Edge-Jetson-Nano* cluster, and it could not serve them. To avoid this, we need to add a component called *circuit breaker* [108] as part of the **Courier**. Once the system detects that a cluster is unavailable or fails in executing a request, the *circuit breaker* gets notified. The algorithm tries to find another suitable cluster instead. After a certain amount of time, the *circuit breaker* checks if the cluster becomes available again. In that case, the cluster can be used again for function delivery. This algorithm can be easily extended to work on other metrics or a combination of metrics as well.

FDN-SLO-Aware: This algorithm adapts the weights of clusters according to the functions' execution time in the target clusters and the defined SLOs. In our evaluation results, we observed that the P90 response time for all the functions when load balanced with this algorithm adhered to the defined SLO of one second. This algorithm performed the best among all the algorithms, providing the lowest response times. When designing this algorithm, we enabled the zero weight policy for the clusters. It means if the execution time on a cluster goes beyond the defined SLO, then the weight of that cluster will be set to zero. We can observe this in the weights graphs of the functions, that certain clusters' weights have been zero. Furthermore, we also observed that the weight assigned to *PVT-CLD-LRZ-OW* cluster for all the functions is zero. It can be attributed to the high cold-start time of a function in the OpenWhisk serverless compute platform. As a result, this algorithm would assign zero weight to it. Requests would only come back to this cluster if no cluster can fulfill the requests under the defined SLOs. Therefore, its weight remains zero for the evaluation time.

FDN-LeastCon: In all the evaluation scenarios, we observed that this algorithm followed a different approach than the FDN-Latency-Aware by sending more invocations to different clusters. For example, in the case of FDN-Latency-Aware algorithm for `nodeinfo` (see Figure 10.15a), we observe that most of the invocations are sent to the *PUB-CLD-AWS* cluster which handled `257.47` MSI/min, while FDN-LeastCon algorithm sent most of the invocations to *PUB-CLD-GCF* cluster, which handled `144.14` MSI/min with MET of `0.003s`. However, in the end, both algorithms still resulted in almost similar P90 response time (Table 10.11). This shows that maintaining the open connections count for each cluster and distributing invocations based on it can also result in a good performance. Additionally, in most cases, when using this

algorithm, most of the invocations go to the *PUB-CLD-GCF* cluster. We suspect, it's due to fewer number of open connections when using GCF serverless compute platform.

Cloud-Centric Algorithms: All the cloud-centric load balancing algorithms (FDN-RR-Cld, FDN-Latency-Aware-Cld, FDN-SLO-Aware-Cld and FDN-LeastCon-Cld) have the lowest response times among all algorithms since they have the advantage of load balancing across high compute capable cloud clusters only. In most cases, FDN-SLO-Aware-Cld has the lowest P90 response times compared to other algorithms. Furthermore, with the increase in user requests (R1 request trace), these algorithms can also adapt and deliver approximately the same response times as those with lower user requests (R2 request trace).

10.7.2 Clusters Performance

We have used six clusters based on four different FaaS platforms (OpenWhisk, OpenFaaS and GCFs, and AWS Lambda). These six clusters were distributed across the edge-cloud continuum, with two public cloud clusters, two private cloud clusters, and two edge clusters. One can easily integrate more clusters within FDN, designed to scale easily with clusters. Edge clusters having low compute resources could not compete with the cloud clusters. However, in some cases like for *gzip-compression* function in Figure 10.19a for FDN-LeastCon algorithm, we can observe that the edge cluster *Edge-Jetson-Nano* (around `227.14` MSI/min) handled more invocations than *PVT-CLD-LRZ-OW* (`216.62` MSI/min). Collaboration between edge and cloud clusters provides a perfect combination to offload the invocations from edge clusters to the cloud clusters. Among cloud clusters, public cloud clusters have the advantage of theoretically unlimited resources as compared to private cloud clusters. In most cases, public cloud clusters have the lowest execution time compared to private cloud clusters. Between the two public serverless compute platforms, in some cases AWS lambda has the lowest execution time (for function `dd`, `linpack`, `nodeinfo`, `primes`) and in some cases Google Cloud Function (for functions `lr-prediction`, `image-processing`). For functions, `sentiment-analysis`, `json-loads`, and `gzip-compression` OpenWhisk based cluster has the lowest time.

In most cases, among the two private cloud clusters, *PVT-CLD-LRZ-OW* performed better than the *PVT-CLD-LRZ-OF* cluster. It shows that the OpenWhisk platform can scale much better than the OpenFaaS platform, since both clusters have the same resources. Furthermore, OpenWhisk use optimized caching and distinguishes between cold, prewarm, and warm containers to address the cold-start problem [202]. Prewarm containers are containers that already have the runtime environment for an action setup. For example, when OpenWhisk's algorithm anticipates Node.js-based actions, it will start preparing generic Node.js containers, which reduces most of the cold-start time. When an action is executed frequently, OpenWhisk will detect that and keep its containers warm. Warm containers are containers where the action is already initialized and ready to be run at any time. On the other hand, OpenFaaS does not have the concept of warm and pre-warm containers; as a result, this can affect the performance of the target cluster when using it. OpenFaaS, like OpenWhisk, supports the option to scale to zero and save money on idle resources. OpenFaaS, on the other hand, provides support for low-end edge devices with ARM processors and therefore is a clear candidate for usage on edge clusters.

10.8 Summary

In this chapter, we presented the evaluation results of five fronts of FDN. First, we discussed the baseline performance of each microbenchmark on different clusters (§9.1.1. We showcased that the performance of the functions when deployed on heterogeneous clusters spread across the edge-cloud continuum can vary drastically, resulting in performance differences. We also presented that FaaS functions use a different

amount of resources (CPU, Memory, Disk, and Network Usage) to execute the task. Depending on internal implementations, this resource usage can vary with different serverless compute platforms. Second, we showed that across all the functions and two workloads, the overhead of FDN is below 5% (§10.10). We also evaluated the correctness of function delivery policies generation based. Results indicate that the FDN ensures that buckets are only replicated to allowed locations and quickly react to changes to keep the FDN's function delivery policies up-to-date. Fourth, we discussed FDN's bucket replication performance where we showed that FDN's bucket replication performance depends entirely on MinIO's `mc` command-line tool and its `mc mirror` directive and the advantage it presents to FDN when doing replications across clusters. Fifth, load balancing results indicate that FDN-SLO-Aware and its cloud variant performed the best among all the algorithms and for all functions (§10.6). Its result always adhered to the defined SLO. Cloud-centric algorithms have the advantage of load balancing across high compute capable cloud clusters. We observed that they have the lowest P90 response time against their normal counterparts. FDN-Latency-Aware and FDN-LeastCon performed similarly and better than the FDN-RR algorithm. We also presented the results of load balancing on the serverless application (`faas-composer`) consisting of multiple functions (§10.6.4).

Edge clusters usually have low compute resources and, therefore, could not compete with the cloud clusters. However, in some cases like for *gzip-compression* function in Figure 10.19a for FDN-LeastCon algorithm, we can observe that the edge cluster *Edge-Jetson-Nano* (around `227.14` MSI/min) handled more invocations than *PVT-CLD-LRZ-OW* (`216.62` MSI/min). Collaboration between edge and cloud clusters provides a perfect combination to offload the invocations from edge clusters to the cloud clusters. Among cloud clusters, public cloud clusters have the lowest execution time compared to private cloud clusters.

11

Conclusion and Future Outlook

"Set your goals high, and don't stop till you get there."

— Bo Jackson

In this chapter, we draw the conclusions from our work in §11.1 and present a future outlook in §11.2.

11.1 Conclusion

Due to the current limitations of serverless computing towards the support of seamless function deployments across the edge-cloud continuum, we introduced an extension to the concept of FaaS as a programming interface for serverless computing across the edge-cloud continuum. This extension is a network of distributed heterogeneous serverless compute clusters spread across the edge-cloud continuum called **Function Delivery Network (FDN)** (Chapter 4). FDN provides seamless integration across the edge-cloud continuum by allowing the user to deploy and invoke the functions across heterogeneous serverless compute clusters in the continuum. In order to integrate several serverless compute clusters, we presented our custom *Virtual Kubelet* provider called *FDN-provider* (§4.1.2). *FDN-provider* acts as federation for multiple serverless compute clusters. Every *Virtual Kubelet* node created using *FDN-provider* acts as a proxy to a serverless compute cluster. *FDN-provider* contains the APIs for managing the functions in different serverless compute platforms. It enables mapping the pod's create or delete requests on virtual worker nodes to the function's create or delete requests on underneath serverless compute clusters. This allows to use a unified `kubectl-based` interface for seamless deployment of the FaaS functions across the continuum. FDN currently supports four serverless compute platforms: AWS Lambda, GCF, OpenWhisk, and OpenFaaS. In order to unify monitoring of multiple serverless compute platforms, we developed a tool called *FDN-Monitor* (§4.2.2). It is deployed as a sidecar to the virtual nodes representing the serverless compute clusters for collecting the data. Currently, serverless compute platforms disregard data locality when scheduling functions, thus causing increased response times and inefficient network traffic, incurring more costs, and potentially crippling SLOs. Therefore, we created *Data Orchestrator* within FDN, which is responsible for

managing the data across the storage backends in clusters within FDN. It is used by the *Courier* component of FDN for creating data-aware delivery policies. *Behave* represents the behavioral modeling of FaaS functions within FDN based on the monitoring data collected by *FDN-Monitor* for characterization of the FaaS functions.

We presented two behavioral models based on the monitoring data collected by *FDN-Monitor* for characterization of the FaaS functions: 1) Functions Performance Model (§5.1), and 2) Function Interaction Model (§5.2). In §5.1, we demonstrated the impact of various configuration parameters on the Function Capacity (FC) for the two serverless compute platforms (AWS Lambda and GCF). The introduced methodology and the tool *FnCapacitor* aim to solve the problem of estimating the FC at a certain deployment configuration. In §5.2 we have shown that neural TPPs effectively model the time and class of function invocations in a serverless application. For this purpose, we introduced *TppFaaS*, a system for creating synthetic serverless applications and using their collected data to train and test neural TPPs. With these datasets, we trained and tested the two TPPs: LogNormMix and TruncNorm. Both models managed to capture the latent temporal dynamics of the different applications. The TPPs performed well for all measures for datasets without cold starts. Here, LogNormMix achieved an accuracy of over `0.94` for most applications. Also, the MAE of TruncNorm's time prediction was below `22ms` for most applications. However, the predictions for the datasets with cold starts were more challenging. Here, TruncNorm achieved a MAE between `200ms` and `750ms` for most applications. The high errors resulted from the high variance of the waitTime, which measures the time an invocation request waits for execution in the internal OpenWhisk system. In addition, LogNormMix's function class prediction performance declined for the cold start datasets. Nevertheless, an accuracy above `0.85` was achieved for most applications, which is still satisfactory.

Courier is used to distribute the incoming function invocations across the serverless compute clusters spread across the edge-cloud continuum (Chapter 6). It consists of two components: *Courier Load Balancer* (§6.2) and *Courier Control Plane* (§6.3). *Courier Load Balancer* is based on HAProxy and is composed of two layers. The first is responsible for delivering the function invocations using two function delivery policies: 1) Function-Aware, and 2) Data-Aware (§6.3.1) to the right clusters. Each function delivery policy selects a subset of the available clusters and employs a load balancer for load balancing the incoming requests to the selected backend clusters. We created two load balancing algorithms: 1) Latency-Aware and 2) SLO-Aware (§6.3.2), for load balancing across the subset of clusters. The *Courier Control Plane* is the brain of the Courier and is responsible for configuring the *Courier Load Balancer* based on the *FDN Inventory Database*, and *FDN Monitoring Data* as the input.

Evaluation results of FDN indicate that it is capable of high-performance function delivering and load balancing (Chapter 10). Across all the functions and two workloads, the overhead of FDN is below 5%. Therefore we conclude that the system overhead is negligible (§10.10). We also evaluated the correctness of function delivery policies generation based on the data collected by *Data Orchestrator*. Results show that, the FDN ensures that buckets are only replicated to allowed locations, reacting to changes to keep the replicated data up-to-date, the replica buckets consistent with user set policies and up-to-date of the FDN's function delivery policies. Furthermore, load balancing results indicate that, FDN-SLO-Aware and its cloud variant performed the best among all the algorithms and for all functions (§10.6). It's result always adhered to the defined SLO. Since cloud-centric algorithms have the advantage of load balancing across high compute capable cloud clusters, we see they have the lowest P90 response time as against their normal counterparts. FDN-Latency-Aware and FDN-LeastCon performed similarly and better than the FDN-RR algorithm. Edge clusters having low compute resources could not compete with the cloud clusters. However, in some cases like for *gzip-compression* function in Figure 10.19a for FDN-LeastCon algorithm, we can observe that the edge cluster *Edge-Jetson-Nano* (around `227.14` MSI/min) handled more invocations than *PVT-CLD-LRZ-OW* (`216.62` MSI/min). Collaboration between edge and cloud clusters provides a perfect combination to offload the invocations from edge clusters to the cloud clusters. Among cloud clusters, public

cloud clusters have the advantage of theoretically unlimited resources as compared to private cloud clusters. In most cases, public cloud clusters have the lowest execution time compared to private cloud clusters.

FDN consists of one external component: *SLAM* tool, to overcome the problem of finding the optimal memory configuration for FaaS functions within a serverless application (Chapter7). *SLAM* uses distributed tracing to detect the relationship among the FaaS functions within a serverless application. By modeling each of them, it estimates the execution time for the application at different memory configurations. Using these estimations, *SLAM* determines the optimal memory configuration for the given serverless application based on the specified SLO requirements and user-specified objectives (minimum cost or minimum execution time). Currently, *SLAM* only supports serverless compute clusters based on AWS Lambda. Evaluation results show that the suggested memory configurations guarantee that more than 95% of requests are completed within the predefined SLOs.

In order to provide reliability across the clusters within FDN, we presented two anomaly detection algorithms for FDN: 1) Online memory leak detection in VMs using *Precog* in §8.1, and 2) Anomalous VMMs detection using *IAD: Indirect Anomaly Detection* in §8.2. The *Precog* algorithm is most relevant for the serverless compute clusters, where the cloud administrator does not have access to the source code or know about the internals of the deployed applications. The performance evaluation showed that the *Precog* could achieve a F1-Score of `0.85` with less than half a second prediction time on the real workloads. The *IAD* algorithm is useful for detecting anomalous VMMs in serverless compute clusters. We compared it against the popular change detection algorithms, which could also be applied to the problem. *IAD* algorithm outperforms all the others on an average across four datasets by `11%` with an average accuracy score of `83.7%`. Both algorithms are easily usable in the cloud environment where the fault-detection time requirement is low and can quickly help in detection of the problem.

To conclude, we hope the methodology adopted within FDN based on serverless computing will drive future research to integrate the edge-cloud continuum.

11.2 Future Outlook

Even though we have created the FDN framework diligently, this does not mean it impedes us from performing further improvements and developing other projects around it. During the development of FDN, many difficulties and challenges have been faced, some of which have not been solved or have simply been circumvented. For this reason, it is possible to enhance the proposed architecture. In this section, we aim to explore some future work contributions that could not be completed as part of this dissertation.

11.2.1 Extension of Virtual Kubelet

FDN uses *Virtual Kubelet*, an open-source Kubernetes kubelet implementation that masquerades as a kubelet to create virtual Kubernetes worker nodes representing the serverless compute clusters. Currently, it supports four serverless compute platforms: AWS Lambda, Google Cloud Function, OpenWhisk, and OpenFaaS. It could be extended to include other public serverless compute platforms, such as Azure Functions, and open-source platforms, such as Knative. Furthermore, currently, FDN has to query *FDN-Monitor* (§4.6) to collect various metrics of functions for decision-making. This could also be integrated within the *Virtual Kubelet* pod metrics. One must implement APIs within *Virtual Kubelet* pods representing the functions to get metrics from *FDN-Monitor* for the representing function and use them as pod metrics. Currently, for all serverless compute platforms, the developed *Virtual Kubelet* implementation only supports Python-based functions. This could be extended to other language runtimes as well.

11.2.2 Energy Efficiency and Power-aware Scheduling on Edge Clusters

Another important factor that FDN can use is providing energy efficiency for certain workloads due to the availability of battery-powered target clusters like edge clusters. We have developed a Prometheus-based exporter [238] for measuring the power consumption of the edge devices and mapping that power consumption to the function instances running on the edge devices. This is also integrated into the *FDN-Monitor* (§4.6. Hence devising the algorithm to use this metric for delivering the function invocations on the low energy consumption target clusters could be added to the FDN. Thus, it would help in saving much energy.

11.2.3 Improvement of the Function Delivering Decision-making Policies

Currently, FDN include Function-Aware (§6.3.1.1) and Data-Aware (§6.3.1.2) based delivering of functions. However, it can be improved to include cold starts. This can be done by keeping a low resource cluster always warm, directing initial function invocations to it, and later using weighted collaboration between other target clusters. Moreover, it is also possible to create a dynamic rule inside the load balancer that checks for the warm target platform and directs the initial function invocations to it, leading to better performance. Function delivery can further be extended to include various aspects such as serverless application workflows [95], where functions are delivered to a cluster where all the other functions related to the overall application workflow are also present, and FaaS function compositions prediction (using the developed *TppFaaS* §5.2.2) to pre-warm the function before it gets invoked.

11.2.4 Improvement of the Load-Balancing Algorithms

Another enhancement that we can bring to the architecture is the improvement of the employed load-balancing algorithms. One of the focuses of this dissertation was the development of a scheduling algorithm that would optimize the execution of functions across the edge-cloud continuum. Even though we tried to devise a great solution, it would have been impossible to create the perfect algorithm. Our algorithms are just a starting point that must be improved steadily with further experiments. We suggest the following two algorithms for future enhancement:

- **Capacity-Aware Load Balancing Algorithm**: In this algorithm, we can use the Function Performance model constructed for each function in Section 5.1 along with the current free capacity of function instances on each cluster. Most serverless platforms have a fixed maximum number of function instances, also called concurrency [10, 79, 44]. For example, AWS has 1000 per user, whereas GCF has 3000 per function. Additionally, self-hosted serverless platforms limit the number of containers one can create due to limited resources. Hence, we can use this information to get each cluster's current free capacity of function instances. In this algorithm, first, we find the FCs of the given function on the subset of clusters using the built Function Performance model (§5.1) based on the given function memory, required SLOs, and current free capacities of the clusters. Afterward, we can determine the maximum FC from all the FCs and, based on this, calculate the weight of each cluster by dividing the maximum FC by the FC on that particular cluster. These values can then be assigned as weights to the clusters.
- **SLO-Aware Cloud-Burst Load Balancing Algorithm**: During the evaluation (§10.6), we noticed that even though private cloud and edge clusters had execution times under the SLOs, most invocations still go to public cloud clusters because they have the lowest execution time. Therefore, it can be

avoided by modifying the FDN-SLO-Aware algorithm that it first sends requests only to private cloud clusters and edge clusters until the execution times are below the defined SLOs. Once the execution times go beyond the SLOs or some percentage of the defined SLOs, some requests burst into the public cloud clusters. It can help to avoid unnecessary costs.

11.2.5 Shim for Function Code for all the Serverless Compute Platforms

Currently, in the FDN, the user has to provide the code for a function for all the serverless compute platforms on which it needs to be deployed. Although most of the logic within the different source codes remains the same, they differ by the input parameters, function handler name, and the returned values. This developed mechanism is still acceptable for environments with no frequent changes within the logic of the function code; otherwise, one has to update the function codes for all the platforms. A possible future work could be to create a shim (such as [40]) that could provide an abstraction on top of the common code required by all the serverless platforms. It would only take common code input from the user and automatically generate the function codes for all the serverless compute platforms.

11.2.6 Creating Serverless Storage Backends for the FDN

FDN is currently only capable of interacting with MinIO storage backends. It is because much of the back-end server logic is specifically dedicated to integrating with MinIO deployments, and the FDN relies on MinIO's command-line client to execute replication tasks. MinIO claims to be compatible with AWS S3 storage technologies and therefore suggests the possibility of an S3-compatible FDN. However, this would still pose challenges for use cases where users might be constrained to non-S3 object-storage technologies. To tackle such use cases and support different storage technologies, FDN's architecture could be extended to include *serverless storage backends*. Different serverless storage backends would be created for different storage technologies, hiding the specificities of the different solutions behind a unified storage API. This strategy would remove code from the FDN's *Data Orchestrator* (§4.2.5) specific to particular storage technologies, making the program more general and diminishing its responsibilities and thus also its points of failure. Such an architecture would, however, make replication tasks more complex, as it would no longer be possible to rely on tools like MinIO's command-line utility for efficient and easy bucket mirroring.

11.2.7 Distributed Anomaly Detection in the FDN

Anomaly detection is vital in IT operations and management, especially in a distributed environment, such as FDN involving heterogeneous clusters. We have developed some anomaly detection algorithms (Chapter 8), but these are more suited for the cloud clusters within the FDN. For detecting anomaly detection on the edge clusters, one needs to either send the collected data from the edge clusters to the cloud or run the anomaly detection on the edge clusters. In the former, sending data to the centralized cloud servers would induce latency and violate data privacy legislations like GDPR. In the latter case, one needs to use the high compute capable edge devices. One solution could be to use distributed anomaly detection algorithms such as those introduced in [319, 57, 74] in the FDN, allowing the utilization of the edge-cloud continuum to the full extent.

Appendices

APPENDIX A

Function Delivery Network Configurations

A.1 FDN Design Configurations Templates

In this section, we present various configuration templates related to FDN. We start with the `FDN-Provider` deployment template within the *Virtual Kubelet* responsible for creating the virtual node and attaching *FDN-Monitor* (§4.2.2) as sidecar to it (§A.1.1). In §A.1.2, we provide the function deployment template responsible for creating a pod within the FDN Kubernetes cluster and mapping it to function creation on the respective serverless compute cluster.

A.1.1 FDN-Provider Deployment Template

For integrating a new cluster, one has to start a virtual-node customized for that cluster using a Kubernetes-based `deployment` configuration file with some command line parameters. The template of the `deployment` configuration file along with the command line parameters is shown in Listing A.1. This template is automatically generated and applied when a new cluster is added and registered in the FDN respectively. This design methodology enables using `kubectl-based` create and delete commands for creating and deleting functions on the respective serverless compute cluster. The `deployment` template consists of three containers: 1) `jaeger-tracing` for tracing of events within the *Virtual Kubelet*, however it is currently not used. 2) `fdn-monitor`, the container responsible for starting the *FDN-Monitor* (§4.2.2) and attaching it as sidecar, 3) `vkubelet`, the container which starts and configures *Virtual Kubelet*. The match labels on lines 10-11 allow the pod to be deployed on the virtual node. For `fdn-monitor` container, we pass all the configuration parameters of the serverless compute platform for which it is responsible for collecting metrics (lines 26-33). Furthermore, we pass the other configuration parameters, such as those related to InfluxDB for storing the monitoring data and MinIO configuration for collecting credentials of GCF serverless compute platform. The `vkubelet` container starts the *Virtual Kubelet* and registers as a virtual node within the Kubernetes cluster. We pass various configuration parameters: node name (line 55), which provider to use and its configuration within the *Virtual Kubelet* (lines 57-59), the serverless compute platform to configure (line 63), the configuration of the platform (lines 64-73) and the MinIO configuration (lines 74-76) for extracting the function code.

Listing A.1: FDN-Provider deployment template, responsible for creating the virtual node and attaching it with *FDN-Monitor* for collecting various metrics.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: vkubelet-fdn-edge-2-openfaas-0
5   labels:
6     cluster: vkubelet-fdn-edge-2-openfaas-0
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      cluster: vkubelet-fdn-edge-2-openfaas-0
12  template:
13    metadata:
14      labels:
15        cluster: vkubelet-fdn-edge-2-openfaas-0
16    spec:
17      containers:
18      - name: jaeger-tracing
19        image: jaegertracing/all-in-one:1.22
20      - name: fdn-monitor
21        image: functiondeliverynetwork/fdn-monitor
22        imagePullPolicy: Always
23        env:
24        - name: CLUSTER_API_GW_ACCESS_TOKEN
25          value: hello
26        - name: AWS_ACCESS_KEY_ID
27          value: "test"
28        - name: AWS_SECRET_ACCESS_KEY
29          value: "test"
30        - name: CLUSTER_AUTH
31          value: "test"
32        - name: CLUSTER_GATEWAY_PORT
33          value: "31112"
34        ...
35        - name: MINIO_ENDPOINT
36          value: "minio:9000"
37        - name: MINIO_ACCESS_KEY
38          value: minio
39        - name: MINIO_SECRET_KEY
40          value: sMGwzbks7sMFTW9Y
41        - name: POWER_COLLECTION
42          value: "True"
43
44        - name: INFLUXDB_ADMIN_TOKEN
45          value: a_secure_admin_token_for_admin_fdn
```

```
46          - name: INFLUXDB_BUCKET
47            value: fdn_monitoring_bucket
48          ...
49        - name: vkubelet
50          image: functiondeliverynetwork/virtual-kubelet:latest
51          imagePullPolicy: Always
52          args:
53          - /virtual-kubelet
54          - --nodename
55          - vkubelet-fdn-edge-2-openfaas-0
56          - --provider
57          - fdn
58          - --provider-config
59          - /vkubelet-fdn-openwhisk-0-cfg.json
60          - --startup-timeout
61          - 10s
62          - --serverless-platform-name
63          - openfaas
64          - --serverless-platform-apihost
65          - "vmschulz43.in.tum.de:31112"
66          - --serverless-platform-auth
67          - "MXfUrseR4mlB"
68          - --serverless-platform-config-bucket
69          - "credentials"
70          - --serverless-platform-config-object
71          - "vkubelet-fdn-edge-2-openfaas-0"
72          - --serverless-platform-region
73          - "europe-west3"
74          - --minio-endpoint
75          - "{{ hostvars[inventory_hostname].ansible_host }}:9000"
76          - --minio-accesskey-id
77          ...
78          env:
79          - name: JAEGER_AGENT_ENDPOINT
80            value: localhost:6831
81          - name: KUBELET_PORT
82            value: "10250"
83          - name: VKUBELET_POD_IP
84          ...
```

A.1.2 FDN-Provider Function Deployment Template

In FDN for creating or deleting functions on the respective serverless compute cluster, one has to create or delete a `pod` on the virtual node mapping that cluster. For creating and deleting a `pod`, we use `kubectl-based` create and delete commands. At first, we create a `deployment` file containing the specification of the pod and its mapping function. It is automatically created by FDN based on the input specified by the

user through the *FDN-UI*. The Kubernetes `deployment` file template for creating a function is shown in Listing A.2. In the file, line 5 tells the Kubernetes the namespace in which the deployment will be created. This namespace corresponds to the serverless compute cluster on which the function needs to be created. We take the container image specified on Line 19 as the function image; one can also specify the generic images specific to the serverless compute platform, for example, `python37` on GCF. Lines 23-26 provide the *Virtual Kubelet* the location of the function code, which it can use to create the function on the respective serverless compute cluster. *Virtual Kubelet* contains the APIs for creating the function on different serverless compute platforms. Lines 27-36 provide information on configuring the function, such as the amount of memory allocated to the function, function timeout, and number of maximum function instances.

Listing A.2: FDN-Provider function deployment template responsible for creating a pod within the FDN Kubernetes cluster and mapping it to function creation on the respective serverless compute cluster.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: gzipcompression
5    namespace: vkubelet-fdn-edge-2-openfaas-0
6    labels:
7      function: gzipcompression
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       function: gzipcompression
13   template:
14     metadata:
15       labels:
16         function: gzipcompression
17     spec:
18       containers:
19        - image: ansjin/gzip-compression:openfaas
20         imagePullPolicy: Always
21         name: gzipcompression
22         env:
23          - name: BUCKET_NAME
24            value: "openfaas"
25          - name: OBJECT_NAME
26            value: "gzip.py"
27          - name: FUNCTION_CPU
28            value: "3000"
29          - name: FUNCTION_MEMORY
30            value: "1024"
31          - name: FUNCTION_TIMEOUT
32            value: "50000"
33          - name: FUNCTION_CONCURRENCY
34            value: "10"
35          - name: FUNCTION_LOGSIZE
```

(a) *infrastructure-centric* metrics

(b) *platform-centric* metrics

(c) load balancer metrics

Figure A.1.: Three different Grafana dashboard within *FDN-Monitor* showcasing various metrics across the clusters.

```
36        value: "80"
37      dnsPolicy: ClusterFirst
38      nodeName: vkubelet-fdn-edge-2-openfaas-0
39      nodeSelector:
40        kubernetes.io/role: agent
41        type: virtual-kubelet
42      tolerations:
43      - key: virtual-kubelet.io/provider
44        operator: Exists
```

A.2 FDN-Components

A.2.1 FDN-Monitor Grafana Dashboards

We have created by default three Grafana dashboards, shown in Figure A.1 representing the metrics from the infrastructure (Figure A.1a), platform (Figure A.1b), and load balancer (Figure A.1c). **Infrastructure-Centric metrics** dashboard in Figure A.1a shows the metrics from the host machines in the cluster. These metrics only exist for clusters hosted on edge or on-premise. The amount and usage over time of static resources, such as the number of cores, memory usage, Disk I/O, and network usage of individual nodes within a cluster, are shown in this dashboard. **Platform-Centric metrics** dashboard in Figure A.1b shows the metrics of the functions from the various serverless compute platforms. Metrics such as the number of function invocations resulting from the received requests, the number of function instances, the execution

(a) Clusters within FDN view.

(b) Adding a new cluster view.

(c) Functions within FDN view.

(d) Adding a new function view.

Figure A.2.: Four different views within *FDN-UI*.

time of the function (excluding the startup latency), etc., are shown in this dashboard. Additionally, for platforms hosted on Kubernetes, we collect the function's resource consumption metrics such as CPU, memory, Disk I/O, and network usage, which are shown as well. **Load balancer metrics** dashboard in Figure A.1c shows the metrics of the *Courier* coming from the **HAProxy**. We mainly use this dashboard to monitor the weights assigned to different clusters for functions within the FDN.

A.2.2 FDN-UI

The UI screenshots showing four different views are presented in Figure A.2. Figure A.2a shows the overview of the clusters which are part of the FDN and assigned to the user. In this view, one can see how many functions are running on a cluster, along with various configuration parameter values of each cluster. Figure A.2b shows the view of registering a new cluster as part of the FDN. In Figure A.2c, we see the overview of the functions created by the user and running in FDN. One can see which clusters the function is running, the function's configuration values, and the load balancing algorithm used for the function. The user can edit, delete, and deploy the function from the same view. Figure A.2d shows the view of adding a new function in the FDN.

A.3 FDN Test Framework

The configuration file used by the evaluation framework (§9.3.2) for creating the scenarios and evaluating the FDN. It contains five sections. The first one specifies the global parameters (lines 1-4): the output directory of evaluation (line 2), the functions' configuration file (line 3), and the name of the test (line4).

The second section specifies the parameters for configuring the load generator (§9.3.2.3) (lines 7-12). Here we specify how long one stage (line 8) is for load testing and how many stages to execute (line 9). The load generator would create VUs based on these configurations. The third section contains all the experiments under the current test (lines 14-24). Here we specify all the functions to evaluate along with the algorithm (represented by cluster name on lines 16, 20, and 23) and trace number (line 17) for the evaluation. The fourth and fifth section provides the configuration parameters for the InfluxDB used by the load generator and the InfluxDB used by the *FDN Monitor*. The evaluation framework uses the *FDN Monitor* InfluxDB for collecting various metrics and storing them as files for later graph creation. In contrast, the other InfluxDB stores the load generation data.

Listing A.3: The configuration file used by the evaluation framework for creating the scenarios and evaluating the FDN.

```
1  global:
2    tests_directory: "./tests"
3    functions_config_file: "config/functions_config.yaml"
4    testName: experiment1
5
6  ## loadGen Settings
7  loadTestSettings:
8    stageDuration: 10s
9    totalStages: 120 # for total test duration to be 120 * 10
10   traces:
11     path: ./traces.csv
12     k6Script: k6/script.js
13
14 experiments:
15 - function_name: func-1-action-primes
16   cluster_name: lrz-cloud-1
17   trace_number: 4
18   ...
19 - function_name: func-1-action-primes
20   cluster_name: fdn-slo
21   trace_number: 4
22 - function_name: func-1-action-primes
23   cluster_name: fdn-rr
24   trace_number: 3
25
26 ## influxdb
27 k6InfluxDB:
28   host: localhost
29   port: 8086
30   db: experiments
31 ## fdn influxdb
32 FDNInfluxDB:
33   host: "fdn.caps.in.tum.de"
34   port: 8086
35   bucket: fdn_monitoring_bucket
36   ..
```

APPENDIX B

Source Code Availability

All the source related to this dissertation exists within the *Function-Delivery-Network* organization of GitHub (`https://github.com/Function-Delivery-Network`). URLs of the individual components presented in this dissertation within the *Function-Delivery-Network* organization are shown in Table B.1

Table B.1.: Source code links related to the components presented in this dissertation.

Category	Component Name	URL
`FDN Internal`	FDN-Monitor	FDN-Monitor
	FDN-Courier	FDN-Courier
	FDN-UI, Cluster Service	FDN-Cluster-Service
	FDN Virtual Kubelet	virtual-kubelet
	FDN Database	FDN-Postgres-DB
`FDN Automation`	FDN Evaluation Framework	FDN-Testing-Framework
	FDN K8s Manifests	FDN-K8s-Manifests
	FDN Edge-Multiple-Boards	IaaC-OpenFaaS-Edge-Multiple-Boards
	FDN Edge-Jetson-Nano	IaaC-OpenFaaS-Edge-Jetson-Nano
	FDN OpenWhisk Pvt. Cloud	IaaC-OpenWhisk-LRZ-Cloud
	FDN OpenFaaS Pvt. Cloud	IaaC-OpenFaaS-LRZ-Cloud
`FDN External`	TppFaaS	TppFaaS
	FaDO for data orchestration	FaDO
	SLAM	SLAM
	FnCapacitor	fnCapacitor
	Memory Leak Detection	memory_leak_detection

APPENDIX C

List of Authored and Co-authored Publications

C.1 Publications Associated with the Dissertation

C.1.1 Journal Articles

- M. Steinbach, A. Jindal, M. M. Chadha. Gerndt and S. Benedict, "TppFaaS: Modeling Serverless Functions Invocations via Temporal Point Processes," in IEEE Access, vol. 10, pp. 9059-9084, 2022, https://doi.org/10.1109/ACCESS.2022.3144078.
- A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, P. Chen. Function delivery network: Extending serverless computing for heterogeneous platforms. Software Practice and Experience. 2021; 51: 1936– 1963. https://doi.org/10.1002/spe.2966
- A. Jindal., M. Gerndt. From DevOps to NoOps: Is It Worth It?. In: Ferguson, D., Pahl, C., Helfert, M. (eds) Cloud Computing and Services Science. CLOSER 2020. Communications in Computer and Information Science, vol 1399. Springer, Cham, 2021. https://doi.org/10.1007/978-3-030-72369-9_8

C.1.2 Conference Articles

- G. Safaryan, A. Jindal, M. Chadha and M. Gerndt, "SLAM: SLO-Aware Memory Optimization for Serverless Applications," 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), 2022, pp. 30-39, https://doi.org/10.1109/CLOUD55607.2022.00019.
- C. P. Smith, A. Jindal, M. Chadha, M. Gerndt and S. Benedict, "FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters," 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC), 2022, pp. 17-25, https://doi.org/10.1109/ICFEC54809.2022.00010.

- T. Zubko, A. Jindal, M. Chadha, M. Gerndt (2022). MAFF: Self-adaptive Memory Optimization for Serverless Functions. In: Montesi, F., Papadopoulos, G.A., Zimmermann, W. (eds) Service-Oriented and Cloud Computing. ESOCC 2022. Lecture Notes in Computer Science, vol 13226. Springer, Cham. https://doi.org/10.1007/978-3-031-04718-3_9
- A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt (2021). Courier: delivering serverless functions within heterogeneous FaaS deployments. In Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC '21). Association for Computing Machinery, New York, NY, USA, Article 11, 1–10. https://doi.org/10.1145/3468737.3494097
- C. Fan, A. Jindal, and M. Gerndt (2020). Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application. In Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER http://dx.doi.org/10.5220/0009792702040215

C.1.3 Workshop Articles

- A. Jindal, M. Chadha, S. Benedict, and M. Gerndt. 2021. Estimating the capacities of function-as-a-service functions. In Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC '21). Association for Computing Machinery, New York, NY, USA, Article 19, 1–8. https://doi.org/10.1145/3492323.3495628
- A. Jindal, I. Shakhat, J. Cardoso, M. Gerndt, V. Podolskiy. (2022). IAD: Indirect Anomalous VMMs Detection in the Cloud-Based Environment. In: Service-Oriented Computing – ICSOC 2021 Workshops. ICSOC 2021. Lecture Notes in Computer Science, vol 13236. Springer, Cham. https://doi.org/10.1007/978-3-031-14135-5_15
- A. Jindal, P. Staab, J. Cardoso, M. Gerndt, V. Podolskiy (2021). Online Memory Leak Detection in the Cloud-Based Infrastructures. In: Service-Oriented Computing – ICSOC 2020 Workshops. ICSOC 2020. Lecture Notes in Computer Science(), vol 12632. Springer, Cham. https://doi.org/10.1007/978-3-030-76352-7_21

C.1.4 Poster

- A. Jindal, M. Chadha, M. Gerndt, J. Frielinghaus, V. Podolskiy and P. Chen, "Poster: Function Delivery Network: Extending Serverless to Heterogeneous Computing," 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), 2021, pp. 1128-1129, https://doi.org/doi:10.1109/ICDCS51616.2021.00120.

C.2 Other Publications

C.2.1 Journal Articles

- V. Podolskiy, A. Jindal, & M. Gerndt (2019). Multilayered autoscaling performance evaluation: Can virtual machines and containers co-scale? International Journal of Applied Mathematics and Computer Science, 29(2), 227–244.

C.2.2 Conference Articles

- S. P. Baller, A. Jindal, M. Chadha and M. Gerndt, "DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices," 2021 IEEE International Conference on Cloud Engineering (IC2E), 2021, pp. 20-30, https://doi.org/10.1109/IC2E52221.2021.00016.
- Thomas van Loo, A. Jindal, M. Chadha, and M. Gerndt. "Scalable Infrastructure for Workload Characterization of Cluster Traces". In Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022), ISBN 978-989-758-570-8, ISSN 2184-5042, pages 254-263.
- L. Espe, A. Jindal., V. Podolskiy and M. Gerndt. (2020). Performance Evaluation of Container Runtimes. In Proceedings of the 10th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER http://dx.doi.org/10.5220/0009340402730281
- A. Jindal, M. Gerndt, M. Bauch and H. Haddouti, "Scalable Infrastructure and Workflow for Anomaly Detection in an Automotive Industry," 2020 International Conference on Innovative Trends in Information Technology (ICITIIT), 2020, pp. 1-6, https://doi.org/10.1109/ICITIIT49094.2020.9071555.
- A. Jindal, V. Podolskiy, and M. Gerndt. 2019. Performance Modeling for Cloud Microservice Applications. In Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE ’19). Association for Computing Machinery, New York, NY, USA, 25–32. https://doi.org/10.1145/3297663.3310309
- A. Jindal, V. Podolskiy, and M. Gerndt, "Multilayered Cloud Applications Autoscaling Performance Estimation," 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2), 2017, pp. 24-31, https://doi.org/10.1109/SC2.2017.12.
- V. Podolskiy, A. Jindal and M. Gerndt, "IaaS Reactive Autoscaling Performance Challenges," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, pp. 954-957, https://doi.org/10.1109/CLOUD.2018.00144.

C.2.3 Workshop Articles

- M. Chadha, A. Jindal, and M. Gerndt. 2020. Towards Federated Learning using FaaS Fabric. In Proceedings of the 2020 Sixth International Workshop on Serverless Computing (WoSC’20). Association for Computing Machinery, New York, NY, USA, 49–54. https://doi.org/10.1145/3429880.3430100
- A. Jindal, V. Podolskiy, and M. Gerndt. 2018. Autoscaling Performance Measurement Tool. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE ’18). Association for Computing Machinery, New York, NY, USA, 91–92. https://doi.org/10.1145/3185768.3186293

Index

Access Control List, 79
Actions, 18
Amazon CloudWatch, 38
Ansible, 37
Application Call Graph Builder, 90
AWS Lambda, 20
AWS Lambda function, 20

Bare metal hypervisor, 8
Behave, 47
Branch and Border, 114
Branch and Border online, 114

cgroup, 11
Change Point, 105, 111
Change Points, 105
Change Points Detector, 111
Cloud Clusters, 123
Cloud computing, 12
Cloud Deployment Models, 13
Cloud service model, 13
Clusters Management, 43
Cold-start, 17
Concurrent instances, 54
Conditional intensity function, 60
Container runtime, 11
containerd, 11
Containers, 11
Courier, 48, 76
Courier Control Plane, 77, 81
Courier Load Balancer, 77, 78
Courier Load Balancer Configuration, 79
CPU virtualization, 9
Critical Time, 104

Data Orchestrator, 44
Data-Aware Delivery Policy, 82
DD, 119
Docker, 11

Edge Cluster, 36
Edge Clusters, 122
Edge computing, 14
Edge-Jetson-Nano, 122
Edge-Multi-Boards, 123
Edge-to-Cloud Continuum, 11
Emulation, 10

F1-Score, 115
FaaS, 1
FaaS application architecture, 22
FaaS Function, 16
FaaS functions, 22
Faas-Composer, 121
faas-idler, 19
faas-netes, 19
faas-provider, 19
FDN Functions Management, 46
FDN Inventory Database, 41
FDN Load balancing, 83
FDN Load Generator, 127
FDN Management Cluster, 34
FDN Monitoring Database, 41
FDN Test Framework, 126
FDN-Latency-Aware, 155
FDN-Latency-Aware-Cld, 155
FDN-Least, 155
FDN-LeastCon-Cld, 155
FDN-Monitor, 38, 40
FDN-provider, 33
FDN-RR, 155
FDN-RR-Cld, 155
FDN-SLO-Aware, 155
FDN-SLO-Aware-Cld, 155

FDN-UI, 49
Firecracker, 11
FnCapacitor, 47, 51
Full virtualization, 9
Function Capacity, 50
Function Code, 20
Function concurrency, 54
Function Delivery Network, 4, 30
Function Delivery Policies, 81
Function initTime, 64
Function Instance, 17
Function Interaction Model, 58
Function Performance Model, 50
Function waitTime, 64
Function-as-a-Service, 15
Function-Aware Delivery Policy, 81
Functions Interaction Model, 47
Functions Performance Model, 47
Functions Sandboxing, 52

Google Cloud Functions, 19
gVisor, 11
Gzip-Compression, 119

HAProxy, 78
Hardware Assisted virtualization, 9
Hazard function, 61
Hosted hypervisor, 9
Hybrid Cloud, 13

I/O virtualization, 10
Image-processing, 120
Image-spec, 11
Indirect Anomaly Detection, 111
Infrastructure-as-a-Service, 13
Infrastructure-Centric metrics, 38
Invocations Traces, 128
Isolation Forest, 115
Isolation Forest Features, 115

Json-loads, 120

K6, 128

Latency-Aware Load Balancing Algorithm, 84
Least-connection, 84
Linear Regression, 53
Linpack, 119
Load balancer, 78
Local Zones, 14
LogNormMix, 62, 66
Lr-prediction, 120

Mean Execution Time, 130
Mean Successful Invocations, 130
Mean-based Detector, 111
Memory leak, 103
Memory virtualization, 9
Microservices application architecture, 21
MicroVMs, 11
MinIO, 44
Monolithic application architecture, 20
Multi-Access Edge Computing, 14
Multi-Cloud, 13

Namespace, 11
Negative Log-Likelihood, 69
Neural Temporal Point Processes, 61
Nodeinfo, 119

OpenFaaS, 19
OpenFaaS Watchdog, 19
OpenTracing, 3
OpenWhisk, 18
OpenWhisk Composer, 58
OpenWhisk Invoker, 18
OS-level virtualization, 10

Para-I/O virtualization, 10
Paravirtualization, 9
Platform-as-a-Service, 14
Platform-Centric metrics, 38
Polynomial Regression, 53
Precog, 104
Primes, 119
Private Cloud, 13
Private Cluster, 36
Private-cloud Cluster, 123
PUB-CLD-AWS, 124
PUB-CLD-GCF, 125
Public Cloud, 13
Public-cloud Cluster, 36, 124
PVT-CLD-LRZ-OF, 123
PVT-CLD-LRZ-OW, 123

Random Forest Regression, 53
Recurrent Marked Temporal Point Processes, 62
Response time, 130

Reverse Proxy, 78
Ridge Regression, 53
Round Robin, 84
Runtime-spec, 11

Sawtooth pattern, 107
Serverless compute cluster, 4
Serverless compute platform, 2, 18
Serverless computing, 1, 2, 14, 15
Single Root I/O Virtualization, 10
SLAM, 88, 89
SLAM-SLO, 93
SLAM-SLO-Min-Cost, 93, 94
SLO-Aware Load Balancing Algorithm, 85
Softmax, 66
Software-as-a-Service, 14
Storage Deployments, 42

Temporal Point Process, 58
Temporal Point Processes, 58
Terraform, 37
TppFaaS, 48, 58, 63
TruncNorm, 67
Type I hypervisors, 8
Type II hypervisors, 8

User-Centric metrics, 38

Virtual Kubelet, 33
Virtual Machine Monitor, 9
Virtual Node, 33
Virtualization, 8
VM-like containers, 11

Weighted Round Robin, 84
wskCLI, 18

Z-score-based Detector, 111

List of Figures

1.1 A schematic overview of the contributions made in this dissertation. 5

2.1 Overview of four isolation methods (VMs, Linux Containers, gVisor-based, and Firecracker-based) for deploying the applications. VMs use a dedicated VMM such as Xen to provide isolation between them. Linux containers use the host kernel's namespace feature to provide isolation between the containers. gVisor-based containers are isolated using the userspace kernel. Firecracker-based MicroVMs use lightweight VMM based on KVM for the isolation. 12
2.2 Typical cloud service models comparison from the aspect of virtualization stack abstraction (y-axis) and focus on business logic (x-axis). Server-based here means that the user or application developer has to configure/manage certain infrastructure parameters. In contrast, the cloud service providers manage infrastructure entirely in serverless computing. 15
2.3 Typical FaaS function invocation procedure. The first time the function is invoked, the serverless compute platform creates an *instance* of the function and runs its *handler method* in it to process the event. When the handler exits or returns a response, it stays active and becomes available to handle other events. 16
2.4 Overview of three different application architectures. 21

4.1 A high-level design of *Virtual Kubelet*. *Virtual Kubelet* is an open-source Kubernetes kubelet implementation that masquerades as a kubelet to connect Kubernetes to other platforms [164]. 32
4.2 A high-level design of *FDN-provider* in *Virtual Kubelet*. Every *Virtual Kubelet* node created using *FDN-provider* acts as a proxy for mapping to actual underneath serverless compute clusters. Pods created/deleted on virtual worker nodes are automatically mapped to functions in the underneath serverless compute clusters. 33
4.3 A high-level architecture design of the Function Delivery Network (FDN). FDN architecture is divided into six layers, with each row in the figure corresponding to a different layer. FDN exposes three different types of APIs to distinguish three different types of clients: user applications that invoke the FaaS function via user APIs, developers, and administrators/operators. All the data within FDN corresponds within the FDN dataplane. 35
4.4 Automation workflow for creating a serverless compute cluster hosted in a private cloud using Terraform and Ansible. 37
4.5 A simplified UML diagram of *FDN-Monitor* showcasing the interfaces for data collection for the different platforms. 40
4.6 Deployment of *FDN-Monitor* as a sidecar with every virtual-node in FDN. Each *FDN-Monitor* instance pulls the metrics from the underneath cluster and aggregate them into InfluxDB. Grafana queries the data from InfluxDB and showcase them in various dashboards. 40
4.7 FDN Inventory data model schema showing different entities as tables and relationships between them. 41
4.8 FDN's Cluster Management workflow showing cluster create/update and delete. 44
4.9 Sequence of events to track a new MinIO deployment. 45

4.10 FDN's Function Management workflow showing function create/update and delete. 47
4.11 FDN's Function invocation workflow. The user requests are received at the *Courier Load Balancer* which selects a subset of clusters based on the set delivering policy, function name and *X-FDN-Bucket* header. The *Courier Load Balancer* then load balances the invocations across the selected subset of serverless compute clusters based on the set load balancing algorithm. 48

5.1 Function Capacitys (FCs) (maximum requests per second) variation with different memory configurations, and function concurrency for AWS Lambda and GCF. 51
5.2 High-level architecture of the *FnCapacitor* and the interaction between its components in a general use case [148]. *FnCapacitor* takes a YAML file as input, and the individual functions from the given application are segregated. These sandboxed functions are then deployed on a serverless compute platform. After the deployment, *FnCapacitor* generates a user workload and repeatedly changes the functions configurations to collect data. The collected metrics data is used for creating the function performance models and are then used for estimating the FCs for different deployment configurations. 52
5.3 `execution_durations` of the sandboxed functions when executed with a load of 50 RPS and no limit on the `function_concurrency`. 54
5.4 `concurrent_instances` of the sandboxed functions for handling the load of 50 RPS with five different memory configurations and no limit on the `function_concurrency`. 55
5.5 FC of the functions when deployed on the two serverless compute platforms for different `function_concurrency` with memory configuration fixed to `256`MB. 56
5.6 Box plot showing the prediction accuracy on the test data across k-folds using DNN model for both serverless compute platforms. 57
5.7 A webshop implemented as a composition of FaaS functions [122]. 58
5.8 The conditional probability density function $f_i^*(t_i)$, the cumulative distribution function $F_i^*(t_i)$, and the survival function $S_i^*(t_i)$ model the time of the next event t_i for a given event history $\mathscr{H}(t_i)$ for a TPP model [88]. 60
5.9 In a neural TPP, the distribution over the next event $P_i(t_i, m_i|\mathscr{H}(t_i))$ is parameterized with the RNN's hidden state vector $\boldsymbol{h}_i$, which encodes the event history $\mathscr{H}(t_i)$ [270]. 62
5.10 *TppFaaS* is a system for modeling serverless applications using TPPs. For this purpose, trace data is collected from synthetic serverless applications that the user can easily create via configuration. The trace data is then used to train a TPP, which models the interactions between the functions in the application. 63
5.11 The owspanprocessor adapts start and endpoint of the original span and adds further attributes. 64
5.12 The owspanattacher adds child spans for waitTime, initTime, and executionTime. 65
5.13 The spans of the invoked functions *f1* and *f2* are mapped to the 3-tuple events e_1 and e_2, which carry the inter-event time τ_i, the function class m_i, and the cold start feature c_i. Given the cold invocation of *f1*, we have $c_1 = 1$. 66
5.14 LogNormMix evaluated via the NLL, with the datasets having no cold starts. A lower value is better, and zero is optimal, except for accuracy, where a higher value is better, and `1.0` is optimal. 71
5.15 TruncNorm evaluation on dataset with no cold starts. The lower value is better, and zero is optimal. A negative value indicates that the predicted time for the invocation was too early. . 72
5.16 LogNormMix evaluated via the NLL, with the datasets having `30%` of the invocations as cold starts. For each application, the TPP is trained and evaluated once with the cold start feature c_i enabled and once with it disabled. A lower value is better, and zero is optimal, except for accuracy, where a higher value is better, and `1.0` is optimal. 73

5.17 TruncNorm evaluation for the dataset having `30%` of the invocations as cold starts. For each application, the TPP is trained and evaluated once with the cold start feature c_i enabled and once with it disabled. The lower value is better, and zero is optimal. A negative value indicates that the predicted time for the invocation was too early. 74

6.1 A high-level overview design of the Courier component of the FDN, responsible for delivering and load balancing user's functions invocations across the edge-cloud continuum in FDN. Courier mainly consists of two components: *Courier Load Balancer* and *Courier Control Plane*. The *Courier Load Balancer* itself consists of two layers. The *Courier Control Plane* is responsible for configuring the *Courier Load Balancer*. 77
6.2 A high-level workflow of the *Courier Control Plane*, responsible for configuring the *Courier Load Balancer* based on various function delivery policies and load balancing algorithms. . 81

7.1 Various factors making it difficult to optimally configure the memory of the FaaS functions. . 88
7.2 High-level architecture of the *SLAM* and the interaction between its components. 89
7.3 Call graphs for the applications used for evaluating *SLAM*. 96
7.4 Actual execution time box plot overlaid with the estimated execution time by *SLAM* run with different SLOs. 97
7.5 Execution time estimation accuracy percentage for the four test applications at different SLOs. 98
7.6 Percentage of the requests conforming to the given SLOs based on the configurations suggested by *SLAM*. 99
7.7 Execution time and the cost when configured with configurations selected by *SLAM* for various objectives. 100
7.8 *SLAM* efficiency and scalability performance 101

8.1 Example memory utilization of a memory leaking VM with the marked anomalous window. 104
8.2 Overall workflow of the *Precog* algorithm. 105
8.3 Algorithm result on three difficult cases having memory leak (a-c) and one not (d). 107
8.4 *Precog*'s prediction method scale linearly. 108
8.5 Insensitive to parameters: *Precog* performs consistently across parameter values. 108
8.6 An example showcasing the propagation of anomalies in a Type-1 hypervisor or VMM to the VMs hosted on it. These anomalies may lead to VMs failures. 109
8.7 Examples showing CPU utilization of two VMs hosted on a VMM. The left sub-figure shows an application running only on VM 2, while the right sub-figure shows the application running on both VMs. We can see a significant decrement in the CPU utilization of the two VMs when an anomaly (high-CPU load) is generated on the VMM (shown by dotted red lines). 111
8.8 High-level system workflow of the implemented system for evaluating IAD algorithm and the interaction between its components in a general use case. 112
8.9 IAD algorithm workflow sequence diagram. 113
8.10 An example profile of an anomalous VMM having 10 VMs in all the datasets used in this work for evaluation. 114
8.11 F1-score variation with the number of VMs corresponding to each algorithm evaluated in this work (§8.2.3.2) and on all the datasets (§8.2.3.1). 116
8.12 Algorithm's detection method scalability with respect to different parameters. 117

9.1 High level workflow of the application used in this work for evaluation. 121
9.2 Schematic high-level diagrams of the two edge clusters based on the embedded devices with limited resources used in this work for FDN's evaluation. 122

9.3 Schematic high-level diagrams of the two private cloud clusters based on two serverless compute platforms used in this work for the FDN's evaluation. 123
9.4 Schematic high-level diagrams of the two public cloud clusters used in this work for FDN's evaluation. 124
9.5 A high-level overview of the various FDN components when deployed on the Kubernetes cluster within the `fdn-related-stuff` namespace. 127
9.6 Workflow of the FDN Test Framework. Its purpose is to test FDN under different scenarios. The configuration file is the input to the framework, containing all the scenarios to execute. The performance metrics data for the scenarios and graphs are the general output of the framework. The client within the framework starts the execution of the load generation by simulating the user workload patterns using the *k6* tool. 128
9.7 Visualization of the function invocation traces in terms of VUs (y-axis) used in this work for FDN evaluation. The x-axis represents the unit time, where one unit time represents `10` seconds. 129

10.1 Plots showing the evaluation results of `nodeinfo` when two *Invocations Traces* (R1 and R2) are used for different clusters. 134
10.2 Plots showing the evaluation results of `primes` when two *Invocations Traces* (R1 and R2) are used for different clusters. 135
10.3 Plots showing the evaluation results of `linpack` when two *Invocations Traces* (R1 and R2) are used for different clusters. 137
10.4 Plots showing the evaluation results of `sentiment-analysis` when two *Invocations Traces* (R1 and R2) are used for different clusters. 138
10.5 Plots showing the evaluation results of `dd` when two *Invocations Traces* (R1 and R2) are used for different clusters. 140
10.6 Plots showing the evaluation results of `gzip-compression` when two *Invocations Traces* (R1 and R2) are used for different clusters. 142
10.7 Plots showing the evaluation results of `json-loads` when two *Invocations Traces* (R1 and R2) are used for different clusters. 143
10.8 Plots showing the evaluation results of `lr-prediction` when two *Invocations Traces* (R1 and R2) are used for different clusters. 145
10.9 Plots showing the evaluation results of `image-processing` when two *Invocations Traces* (R1 and R2) are used for different clusters. 146
10.10 Percentage overhead introduced in terms of P90 response time by FDN when compared against the direct approach for **nodeinfo** function across all the clusters and at two user workload invocations (R1 and R2). One can see that across all the clusters, the overhead is below 5%. 151
10.11 Mirroring different data amounts from the `PVT-CLD-LRZ-OF` to `PVT-CLD-LRZ-OW` using `mc` and without it. 154
10.12 The average and 90th percentile response times of the invocations load balanced using eight different algorithms to the `nodeinfo` function. The results are shown for both *Invocation Traces*. 156
10.13 Details on how the successful invocations made using `Trace R2` to `nodeinfo` function are distributed across each cluster along with their execution times using different load balancing algorithms. 157
10.14 Weights distribution among different clusters during evaluation test for `nodeinfo` function when load balanced using different algorithms for two *Invocation Traces*. 158

10.15 Details on how the successful invocations made using `Trace R1` to `nodeinfo` function are distributed across each cluster, along with their execution times using different load balancing algorithms. 161
10.16 The average and 90th percentile of the response times of the invocation requests, load balanced using eight different algorithms to the `gzip-compression` function. The results are shown for two *Invocation Traces*. 163
10.17 Details on how the successful invocations made using `Trace R2` to `gzip-compression` function are distributed across each cluster, along with their execution times using different load balancing algorithms. 164
10.18 Weights distribution among different clusters during evaluation test for `gzip-compression` function when load balanced using different algorithms for two *Invocation Traces*. 165
10.19 Details on how the successful invocations made using `Trace R1` to `gzip-compression` function are distributed across each cluster, along with their execution times using different load balancing algorithms. 166
10.20 The average and 90th percentile response times of the invocations load balanced using eight different algorithms to the `lr-prediction` function. The results are shown for both *Invocation Traces*. 168
10.21 Details on how the successful invocations made using `Trace R2` to `lr-prediction` function are distributed across each cluster, along with their execution times using different load balancing algorithms. 169
10.22 Weights distribution among different clusters during evaluation test for `lr-prediction` function when load balanced using different algorithms for two *Invocation Traces*. 169
10.23 Details on how the successful invocations made using `Trace R1` to `lr-prediction` function are distributed across each cluster, along with their execution times using different load balancing algorithms. 171
10.24 The average and 90th percentile of the response times of the invocation requests, load balanced using four different algorithms to the `faas-composer` function. The request traces are lowered-down versions of the original ones, with the maximum number of requests per second as 10. 172
10.25 Details on how the successful invocations load tested using lowered-down version of `Trace R1` to `faas-composer` following different algorithms are distributed across the functions and clusters. The rows show the different algorithms (From top to bottom: *FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon) and columns show different functions. 173
10.26 Details on how the successful invocations' execution time load tested using lowered-down version of `Trace R1` to `faas-composer` application following different algorithms are distributed across the functions and clusters. The rows show the different algorithms (From top to bottom: *FDN-RR*, FDN-Latency-Aware, FDN-SLO-Aware and FDN-LeastCon) and columns show different functions. 174
10.27 Weights distribution among different clusters during evaluation test for `faas-composer` function when load balanced using different algorithms for two *Invocation Traces*. 176

A.1 Three different Grafana dashboard within *FDN-Monitor* showcasing various metrics across the clusters. 193
A.2 Four different views within *FDN-UI*. 194

List of Tables

4.1 The summary of the monitoring metrics from *Platform-Centric* and *Infrastructure-Centric* categories for all the four serverless platforms considered in this work, along with the name used by the *FDN-Monitor*. For all these metrics, the data is collected per unit of time. For platforms hosted on Kubernetes, the functions run as pods; therefore, we can also collect the resources' consumption metrics of pods. 39

5.1 Comparison of accuracy results (R^2 score) for estimated FCs for the different approaches. . . 57
5.2 Symbols and their definitions used in the context of building *Functions Interaction Model* . . 59

7.1 Symbols and definitions used in the context of *SLAM* tool. 90

8.1 Symbols and their definitions used in the context of memory leak detection. 103
8.2 Synthetically generated timeseries for each memory leak pattern and their F1-Scores. 106
8.3 Symbols and their definitions used in the context of indirectly detecting anomalous VMMs in a cloud-based environment. 110
8.4 Datasets used in this work for evaluating the algorithms. 114
8.5 The details of the algorithms used in this work for evaluation, along with their input dimension and parameters. 115
8.6 F1-score corresponding to each algorithm evaluated in this work (§8.2.3.2) and on all the datasets (§8.2.3.1). 115

9.1 Summary of the FaaS functions microbenchmarks used as part of this work for evaluating FDN. 120
9.2 Different target heterogeneous clusters spread across edge-cloud continuum used for evaluating the FDN. 125
9.3 An extract of the dataset containing the number of visits to four Wikipedia pages on different dates. Each row represents a Wikipedia page, while the columns correspond to the date of the period from 2015 to 2019. Each cell contains the number of visitors for the given page on the date. In our tests, each day corresponds to a period of `10` seconds. The number of visitors is used as the number of function invocations. However, we scaled down the number of invocations so that the maximum number of VUs is less than `200`. 129

10.1 Three Platform-Centric metrics showing the mean usage of the resources by `nodeinfo` function on different clusters when handling the two *Invocations Traces*. 134
10.2 Three Platform-Centric metrics showing the mean usage of the resources by `primes-python` function on different clusters when handling the two *Invocations Traces*. 136
10.3 Platform-Centric metrics showing the mean usage of the resources by `linpack` function on different clusters when handling the two *Invocations Traces*. 138
10.4 Platform-Centric metrics showing the mean usage of the resources by `sentiment-analysis` function on different clusters when handling the two *Invocations Traces*. 139

10.5 Platform-Centric metrics showing the mean usage of the resources by `dd` function on different clusters when handling the two *Invocations Traces*. 141
10.6 Platform-Centric Metrics showing the mean usage of the resources by `gzip-compression` function on different clusters when handling the two *Invocations Traces*. 142
10.7 Platform-Centric metrics showing the mean usage of the resources by `json-loads` function on different clusters serving the two *Invocations Traces*. 144
10.8 Platform-Centric metrics showing the mean usage of the resources by `lr-Prediction` function on different clusters serving the two *Invocations Traces*. 145
10.9 Platform-Centric Metrics showing the mean usage of the resources by `image-processing` function on different clusters serving the two *Invocations Traces*. 147
10.10 Summary of the average number of invocations made per minute by each cluster for each FaaS function when load tested with two `Invocation Traces`. 148
10.11 Summary of the load balancing algorithms results on the FaaS functions. 179

B.1 Source code links related to the components presented in this dissertation. 196

List of Algorithms

1 Latency-Aware Load Balancing Algorithm 85
2 SLO-Aware Load Balancing Algorithm 86
3 SLAM-SLO Algorithm 94

List of Listings

4.1 An example of JSON input recevied by the cluster register API endpoint for registering the cluster within FDN 43

6.1 An example configuration of *Couier Load Balancer* based on HAProxy. The `frontend` specifies the address on which the server is listening and a set of rules for requests forwarding. The `acl` tag defines a path-based routing policy for sorting the incoming requests to the corresponding backend. Each `backend` specifies the load balancing algorithm used and the set of the clusters that will receive the requests. 80

6.2 An example of a Function-Aware Delivery Policy created for `function_1` which is deployed on `cluster_1, cluster_2 and cluster_4`. 82

6.3 An example of a Data-Aware Delivery Policy created for `function_1` which is deployed on `cluster_1, cluster_2`. Storage bucket `bucket_1` exists on `cluster_1` and storage bucket `bucket_2` exists on `cluster_1, and cluster_2` 83

10.1 Configuration file of *Courier Load Balancer* showing no backends apart from the default backend. 152

10.2 Configuration file of *Courier Load Balancer* showing function invocations requiring the bucket *rep-policy-demo* solely go to the `PVT-CLD-LRZ-OW`. 152

10.3 Configuration file of *Courier Load Balancer* showing three clusters to which function invocations requiring the bucket *rep-policy-demo* solely go to, after increase in number of replica count of the bucket. 152

10.4 Configuration file of *Courier Load Balancer* showing two additional backends after a test function *nodeinfo* is deployed; one for the function and one matching both function and bucket 153

A.1 FDN-Provider deployment template, responsible for creating the virtual node and attaching it with *FDN-Monitor* for collecting various metrics. 190

A.2 FDN-Provider function deployment template responsible for creating a pod within the FDN Kubernetes cluster and mapping it to function creation on the respective serverless compute cluster. 192

A.3 The configuration file used by the evaluation framework for creating the scenarios and evaluating the FDN. 195

Acronyms

ACL	Access Control List.
AKS	Azure Kubernetes Service.
AMI	Amazon Machine Image.
API	Application Programming Interface.
ARN	Amazon Resource Name.
AWS	Amazon Web Services.
BaaS	Backend-as-a-Service.
BnB	Branch and Border.
BnBO	Branch and Border Online.
CNCF	Cloud Native Computing Foundation.
CRM	Customer Relationship Management.
CSP	Cloud Service Provider.
DBMS	Database Management System.
DMA	Direct Memory Access.
DNN	Deep Neural Network.
EC2	Elastic Compute Cloud.
eCDF	empirical Cumulative Distribution Function.
EKS	Amazon Elastic Kubernetes Service.
FaaS	Function-as-a-Service.
FC	Function Capacity.
FDaaS	Function-Delivery-as-a-Service.
FDN	Function Delivery Network.
FDN-LeastCon-Cld	FDN Least Connections Algorithm Cloud Only.
FDN-SLO-Aware-Cld	FDN SLO-Aware Algorithm Cloud Only.
FDN-Latency-Aware-Cld	FDN Latency-Aware Algorithm Cloud Only.
FDN-RR-Cld	FDN Round-Robin Algorithm Cloud Only.
FDN-LeastCon	FDN Least Connections Algorithm.
FDN-SLO-Aware	FDN SLO-Aware Algorithm.
FDN-Latency-Aware	FDN Latency-Aware Algorithm.
FDN-RR	FDN Round-Robin Algorithm.
FPGA	Field-Programmable Gate Array.

GCE	Google Compute Engine.
GCF	Google Cloud Function.
GCP	Google Cloud Platform.
GPU	Graphics Processing Unit.
GRU	Gated Recurrent Unit.
HBA	Host Bus Adapter.
HCL	HashiCorp Configuration Language.
HPC	High-Performance Computing.
IaaS	Infrastructure-as-a-Service.
IAD	Indirect Anomaly Detection.
IF	Isolation Forest.
IFF	Isolation Forest Features.
IOMMU	Input–Output Memory Management Unit.
IoT	Internet of Things.
JSON	JavaScript Object Notation.
KVM	Kernel-Based Virtual Machine.
LRZ	Leibniz-Rechenzentrum.
LTE	Long- Term Evolution.
MAE	Mean Absolute Error.
MEC	Multi-Access Edge Computing.
MET	Mean Execution Time.
ML	Machine Learning.
MMU	Memory Management Unit.
MOC	Minimum Overall Cost.
MOET	Minimum Overall Execution Time.
MR-IOV	Multiple Root I/O Virtualization and Sharing Specification.
MSI/min	Mean Successful Invocations per minute.
NATS	Neural Autonomic Transport System.
NEG	Network Endpoint Group.
NFS	Network File System.
NIC	Network Interface Card.
NLL	Negative Log-Likelihood.
NLTK	Natural Language Toolkit.
OCI	Open Containers Initiative.
OS	Operating System.
PaaS	Platform-as-a-Service.
PCI	Peripheral Component Interconnect.
PCIe	Peripheral Component Interconnect Express.

PF	Physical Function.
PV	Persistent Volume.
PVC	Persistent Volume Claim.
ReLU	Rectified Linear Unit.
REST	Representational State Transfer.
RMTPPs	Recurrent Marked Temporal Point Processes.
RNN	Recurrent Neural Network.
RPC	Remote Procedure Call.
RPS	Requests per Second.
RR	Round-Robin.
RTT	Round Trip Time.
SaaS	Software-as-a-Service.
SLO	Service-Level Objective.
SNS	Simple Notification Service.
SQS	Simple Queue Service.
SR-IOV	Single Root I/O Virtualization and Sharing Specification.
SSL	Secure Sockets Layer.
TLB	Translation Lookaside Buffer.
TPP	Temporal Point Process.
TPPs	Temporal Point Processes.
TPU	Tensor Processing Unit.
URL	Uniform Resource Locator.
VCS	Version Control System.
VF	Virtual Function.
vHBA	virtual Host Bus Adapter.
VM	Virtual Machine.
VMM	Virtual Machine Monitor.
vNIC	virtual Network Interface Card.
VU	Virtual User.
WRR	Weighted Round-Robin.
YAML	YAML Ain’t Markup Language.

Bibliography

[5] Alexandru Agache et al. “Firecracker: Lightweight virtualization for serverless applications”. In: *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 2020, pp. 419–434.

[6] Nabeel Akhtar et al. “COSE: Configuring Serverless Functions using Statistical Learning”. In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2020, pp. 129–138. DOI: 10.1109/INFOCOM41043.2020.9155363.

[7] Istemi Ekin Akkus et al. “SAND: Towards High-Performance Serverless Computing”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[8] Zaid Al-Ali et al. “Making Serverless Computing More Serverless”. In: July 2018, pp. 456–459. DOI: 10.1109/CLOUD.2018.00064.

[9] Omid Alipourfard et al. “Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics”. In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI’17. USENIX Association, 2017, 469–482. ISBN: 9781931971379. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard.

[12] Charles Anderson. “Docker [software engineering]”. In: *Ieee Software* 32.3 (2015), pp. 102–c3.

[15] Dimitris Apostolou, Yiannis Verginadis, and Gregoris Mentzas. “In the Fog: Application Deployment for the Cloud Continuum”. In: *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*. 2021, pp. 1–7. DOI: 10.1109/IISA52424.2021.9555532.

[17] Austin Aske and Xinghui Zhao. “Supporting Multi-Provider Serverless Computing on the Edge”. In: *Proceedings of the 47th International Conference on Parallel Processing Companion*. ICPP ’18. Association for Computing Machinery, 2018, pp. 1–6. ISBN: 9781450365239. DOI: 10.1145/3229710.3229742. URL: https://doi.org/10.1145/3229710.3229742.

[18] S. M. A. Ataallah, S. M. Nassar, and E. E. Hemayed. “Fault tolerance in cloud computing - survey”. In: *2015 11th International Computer Engineering Conference (ICENCO)*. Dec. 2015, pp. 241–245. DOI: 10.1109/ICENCO.2015.7416355.

[46] Ataollah Fatahi Baarzi et al. “On Merits and Viability of Multi-Cloud Serverless”. In: *Proceedings of the ACM Symposium on Cloud Computing*. New York, NY, USA: Association for Computing Machinery, 2021, 600–608. ISBN: 9781450386388. URL: https://doi.org/10.1145/3472883.3487002.

[47] Ioana Baldini et al. “Serverless computing: Current trends and open problems”. In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20. arXiv: 1706.03178v1.

[48] Daniel Balouek-Thomert et al. "Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows". In: *The International Journal of High Performance Computing Applications* 33.6 (Sept. 2019), pp. 1159–1174. DOI: 10.1177/1094342019877383. eprint: https://doi.org/10.1177/1094342019877383. URL: https://doi.org/10.1177/1094342019877383.

[49] Luciano Baresi, Danilo Filgueira Mendonça, and Martin Garriga. "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture". In: *Service-Oriented and Cloud Computing*. Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen. Springer International Publishing. Cham: Springer International Publishing, 2017, pp. 196–210. ISBN: 978-3-319-67262-5.

[50] Sasa Baskarada, Vivian Nguyen, and Andy Koronios. "Architecting Microservices: Practical Opportunities and Challenges". In: *Journal of Computer Information Systems* (Sept. 2018), pp. 1–9. DOI: 10.1080/08874417.2018.1520056.

[51] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing". In: *Future generation computer systems* 28.5 (2012), pp. 755–768.

[52] David Bermbach, Ahmet Serdar Karakaya, and Simon Buchholz. "Using application knowledge to reduce cold starts in FaaS services". In: *Proceedings of the ACM Symposium on Applied Computing* (2020), pp. 134–143. DOI: 10.1145/3341105.3373909.

[53] David Bermbach et al. "Towards Auction-Based Function Placement in Serverless Fog Platforms". In: Apr. 2020, pp. 25–31. DOI: 10.1109/ICFC49376.2020.00012.

[55] Kashif Bilal et al. "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers". In: *Computer Networks* 130 (2018), pp. 94–120. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.10.002. URL: https://www.sciencedirect.com/science/article/pii/S1389128617303778.

[56] Carl Boettiger. "An introduction to Docker for reproducible research, with examples from the R environment". In: *CoRR* abs/1410.0846 (2014). arXiv: 1410.0846. URL: http://arxiv.org/abs/1410.0846.

[57] Joel W. Branch et al. "In-network outlier detection in wireless sensor networks". In: *Knowledge and Information Systems* 34.1 (2013), pp. 23–54. DOI: 10.1007/s10115-011-0474-5. URL: https://doi.org/10.1007/s10115-011-0474-5.

[58] Antonio Brogi et al. "SeaClouds". In: *ACM SIGSOFT Software Engineering Notes* 39.1 (Feb. 2014), pp. 1–4. DOI: 10.1145/2557833.2557844.

[60] Rajkumar Buyya and Satish Narayana Srirama. *Fog and edge computing: principles and paradigms*. John Wiley & Sons, 2019.

[61] Rajkumar Buyya and Satish Narayana Srirama. "Internet of Things (IoT) and New Computing Paradigms". In: *Fog and Edge Computing: Principles and Paradigms*. 2019, pp. 1–23. DOI: 10.1002/9781119525080.ch1.

[62] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation Computer Systems* 25.6 (2009), pp. 599–616. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2008.12.001. URL: https://www.sciencedirect.com/science/article/pii/S0167739X08001957.

[65] Tyler Caraza-Harter and Michael M Swift. "Blending containers and virtual machines: a study of firecracker and gVisor". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2020, pp. 101–113.

[66] Joao Carreira et al. "Cirrus: A serverless framework for end-to-end ml workflows". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 13–24.

[67] Rich Caruana, Steve Lawrence, and C Lee Giles. "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping". In: *Advances in neural information processing systems*. 2001, pp. 402–408.

[70] Paul Castro et al. "The Rise of Serverless Computing". In: *Commun. ACM* 62.12 (Nov. 2019), 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454. URL: https://doi.org/10.1145/3368454.

[71] Mohak Chadha, Anshul Jindal, and Michael Gerndt. "Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021, pp. 478–483. DOI: 10.1109/CLOUD53861.2021.00062.

[72] Mohak Chadha, Anshul Jindal, and Michael Gerndt. "Towards Federated Learning Using FaaS Fabric". In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC'20. Association for Computing Machinery. Association for Computing Machinery, 2020, 49–54. ISBN: 9781450382045. DOI: 10.1145/3429880.3430100. URL: https://doi.org/10.1145/3429880.3430100.

[73] Narsimha Reddy Challa. "Hardware based i/o virtualization technologies for hypervisors, configurations and advantages-a study". In: *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE. 2012, pp. 1–5.

[74] P. Chen, S. Yang, and J. A. McCann. "Distributed Real-Time Anomaly Detection in Networked Industrial Sensing Systems". In: *IEEE Transactions on Industrial Electronics* 62.6 (2015), pp. 3832–3842.

[75] Qian Chen et al. "On state of the art in virtual machine security". In: *2012 Proceedings of IEEE Southeastcon*. 2012, pp. 1–6. DOI: 10.1109/SECon.2012.6196905.

[76] X. Chen et al. "Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing". In: *IEEE/ACM Transactions on Networking* 24.5 (2016), pp. 2795–2808.

[77] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference* (2014), pp. 1724–1734. DOI: 10.3115/v1/d14-1179. arXiv: 1406.1078.

[83] Eli Cortez et al. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, 153–167. ISBN: 9781450350853. DOI: 10.1145/3132747.3132772. URL: https://doi.org/10.1145/3132747.3132772.

[84] Rodrigo Crespo-Cepeda et al. "Challenges and Opportunities of Amazon Serverless Lambda Services in Bioinformatics". In: *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. BCB '19. Niagara Falls, NY, USA: Association for Computing Machinery, 2019, 663–668. ISBN: 9781450366663. DOI: 10.1145/3307339.3343462. URL: https://doi.org/10.1145/3307339.3343462.

[86] Michael Cusumano. "Cloud computing and SaaS as new computing platforms". In: *Communications of the ACM* 53.4 (2010), pp. 27–29.

[87] Andrea Damiani et al. "BlastFunction: A Full-Stack Framework Bringing FPGA Hardware Acceleration to Cloud-Native Applications". In: *ACM Trans. Reconfigurable Technol. Syst.* 15.2 (Jan. 2022). ISSN: 1936-7406. DOI: 10.1145/3472958. URL: https://doi.org/10.1145/3472958.

[88] A De, U Upadhyay, and M Gomez-Rodriguez. *Lecture Notes for Human-Centered ML: Temporal Point Processes*. 2019.

[90] P. Di Francesco, P. Lago, and I. Malavolta. "Migrating Towards Microservice Architectures: An Industrial Survey". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. Apr. 2018, pp. 29–2909. DOI: 10.1109/ICSA.2018.00012.

[91] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions". In: *Journal of Network and Computer Applications* 103 (2018), pp. 1–17. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.12.001. URL: https://www.sciencedirect.com/science/article/pii/S1084804517303971.

[92] Yaozu Dong, Zhao Yu, and Greg Rose. "SR-IOV Networking in Xen: Architecture, Design and Implementation." In: *Workshop on I/O Virtualization*. Vol. 2. 2008.

[93] Nan Du et al. "Recurrent Marked Temporal Point Processes: Embedding Event History to Vector". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, 1555–1564. ISBN: 9781450342322. DOI: 10.1145/2939672.2939875. URL: https://doi.org/10.1145/2939672.2939875.

[95] Simon Eismann et al. "A Review of Serverless Use Cases and their Characteristics". In: *CoRR* abs/2008.11110 (2020). arXiv: 2008.11110. URL: https://arxiv.org/abs/2008.11110.

[96] Simon Eismann et al. "Serverless Applications: Why, When, and How?" In: *CoRR* abs/2009.08173 (2020). arXiv: 2009.08173. URL: https://arxiv.org/abs/2009.08173.

[97] Simon Eismann et al. "Sizeless: Predicting the Optimal Size of Serverless Functions". In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, 248–259. ISBN: 9781450385343. DOI: 10.1145/3464298.3493398. URL: https://doi.org/10.1145/3464298.3493398.

[98] Adam Eivy. "Be Wary of the Economics of "Serverless" Cloud Computing". In: *IEEE Cloud Comput.* 4.2 (2017), pp. 6–12. DOI: 10.1109/MCC.2017.32. URL: https://doi.org/10.1109/MCC.2017.32.

[99] Tarek Elgamal et al. "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement". In: *CoRR* abs/1811.09721 (2018). arXiv: 1811.09721. URL: http://arxiv.org/abs/1811.09721.

[101] Ron Emerick. "PCI Express IO Virtualization Overview". In: *SNIA Education* (2012).

[102] Lennart Espe. et al. "Performance Evaluation of Container Runtimes". In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,* INSTICC. SciTePress, 2020, pp. 273–281. ISBN: 978-989-758-424-4. DOI: 10.5220/0009340402730281.

[103] Erwin van Eyk et al. "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures". In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, 1–4. ISBN: 9781450354349. DOI: 10.1145/3154847.3154848. URL: https://doi.org/10.1145/3154847.3154848.

[104] Youssef Fahim et al. "Load Balancing in Cloud Computing Using Meta-Heuristic Algorithm." In: *Journal of Information Processing Systems* 14.3 (2018).

[105] Chen-Fu Fan., Anshul Jindal., and Michael Gerndt. "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application". In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,* INSTICC. SciTePress, 2020, pp. 204–215. ISBN: 978-989-758-424-4. DOI: 10.5220/0009792702040215.

[106] Charles Ferrari et al. "Edge Computing for Communication Service Providers: A Review on the Architecture, Ownership and Governing Models". In: *2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2021, pp. 1–6. DOI: 10.23919/SoftCOM52868.2021.9559056.

[107] Kamil Figiela et al. "Performance evaluation of heterogeneous cloud functions". In: *Concurrency and Computation: Practice and Experience* 30.23 (2018), e4792.

[109] Geoffrey C. Fox et al. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research". In: (Aug. 2017). DOI: 10.13140/RG.2.2.15007.87206. arXiv: 1708.08028 [cs.DC]. URL: http://arxiv.org/abs/1708.08028http://dx.doi.org/10.13140/RG.2.2.15007.87206.

[110] Geoffrey C. Fox et al. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research". In: *arXiv e-prints*, arXiv:1708.08028 (Aug. 2017), arXiv:1708.08028. arXiv: 1708.08028 [cs.DC].

[111] Joshua D Gagliardi and Timothy S Munger. *Content delivery network*. https://portal.unifiedpatents.com/patents/patent/US-8868737-B2. US Patent 7,962,580. 2011.

[112] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[114] Joel Gibson et al. "Benefits and challenges of three cloud computing service models". In: *2012 Fourth International Conference on Computational Aspects of Social Networks (CASoN)*. 2012, pp. 198–205. DOI: 10.1109/CASoN.2012.6412402.

[115] M. K. Gokhroo, M. C. Govil, and E. S. Pilli. "Detecting and mitigating faults in cloud computing environment". In: *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*. Feb. 2017, pp. 1–9. DOI: 10.1109/CIACT.2017.7977362.

[122] Martin Grambow et al. "BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms". In: *CoRR* abs/2102.12770 (2021). arXiv: 2102.12770. URL: https://arxiv.org/abs/2102.12770.

[123] J. R. Gunasekaran et al. "Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 199–208. DOI: 10.1109/CLOUD.2019.00043.

[124] Harshit Gupta et al. "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments". In: *Software: Practice and Experience* 47.9 (June 2017), pp. 1275–1296. DOI: 10.1002/spe.2509. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2509. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509.

[127] Joseph M. Hellerstein et al. "Serverless computing: One step forward, two steps back". In: *CIDR 2019 - 9th Biennial Conference on Innovative Data Systems Research* 3 (2019). arXiv: arXiv:1812.03651v1.

[128] Scott Hendrickson et al. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[129] Jiangshui Hong et al. “An Overview of Multi-cloud Computing”. In: *Web, Artificial Intelligence and Network Applications*. Ed. by Leonard Barolli et al. Cham: Springer International Publishing, 2019, pp. 1055–1068. ISBN: 978-3-030-15035-8.

[131] Bryan Hooi and Christos Faloutsos. “Branch and Border: Partition-Based Change Detection in Multivariate Time Series”. In: *SDM*. 2019.

[138] N. Jain and S. Choudhary. “Overview of virtualization in cloud computing”. In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. Mar. 2016, pp. 1–4. DOI: 10.1109/CDAN.2016.7570950.

[139] B. Jambunathan and K. Yoganathan. “Architecture Decision on using Microservices or Serverless Functions with Containers”. In: *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*. Mar. 2018, pp. 1–7. DOI: 10.1109/ICCTCT.2018.8551035.

[140] Nicholas A. James and David S. Matteson. *ecp: An R Package for Nonparametric Multiple Change Point Analysis of Multivariate Data*. 2013. arXiv: 1309.3295 [stat.CO].

[142] Kris Jamsa. *Cloud computing: SaaS, PaaS, IaaS, virtualization, business models, mobile, security and more*. Jones & Bartlett Publishers, 2012.

[143] Jananie Jarachanthan et al. “Astra: Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 756–765. DOI: 10.1109/IPDPS49936.2021.00085.

[146] Anshul Jindal and Michael Gerndt. “From DevOps to NoOps: Is It Worth It?” In: *Cloud Computing and Services Science*. Ed. by Donald Ferguson, Claus Pahl, and Markus Helfert. Cham: Springer International Publishing, 2021, pp. 178–202. ISBN: 978-3-030-72369-9.

[147] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. “Performance modeling for cloud microservice applications”. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE ’19. Mumbai, India: Association for Computing Machinery, 2019, pp. 25–32. ISBN: 9781450362399. DOI: 10.1145/3297663.3310309. URL: https://doi.org/10.1145/3297663.3310309.

[148] Anshul Jindal et al. “Estimating the Capacities of Function-as-a-Service Functions”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC ’21 Companion. New York, NY, USA: Association for Computing Machinery, 2021. ISBN: 978-1-4503-9163-4/21/12. DOI: 10.1145/3492323.3495628. URL: https://doi.org/10.1145/3492323.3495628.

[149] Anshul Jindal et al. “Function delivery network: Extending serverless computing for heterogeneous platforms”. In: *Software: Practice and Experience* 51.9 (2021), pp. 1936–1963. ISSN: 1097024X. DOI: https://doi.org/10.1002/spe.2966. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2966. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966.

[150] Anshul Jindal et al. “Poster: Function Delivery Network: Extending Serverless to Heterogeneous Computing”. In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 2021, pp. 1128–1129. DOI: 10.1109/ICDCS51616.2021.00120.

[151] Anshul Jindal et al. “Scalable Infrastructure and Workflow for Anomaly Detection in an Automotive Industry”. In: *2020 International Conference on Innovative Trends in Information Technology (ICITIIT)*. 2020, pp. 1–6. DOI: 10.1109/ICITIIT49094.2020.9071555.

[152] Eric Jonas et al. “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019).

[153] Eric Jonas et al. “Occupy the cloud”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, Sept. 2017, 445–451. ISBN: 9781450350280. DOI: 10.1145/3127479.3128601. URL: https://doi.org/10.1145/3127479.3128601.

[154] Zhu Kai et al. “Building a private cloud platform based on open source software OpenStack”. In: *2020 International Conference on Big Data and Social Sciences (ICBDSS)*. 2020, pp. 84–87. DOI: 10.1109/ICBDSS51270.2020.00027.

[155] Michael J Kavis. *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014.

[156] D. Kelly, F. Glavin, and E. Barrett. “Serverless Computing: Behind the Scenes of Major Platforms”. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020, pp. 304–312. DOI: 10.1109/CLOUD49709.2020.00050.

[157] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014), pp. 1–15. arXiv: 1412.6980.

[158] Avi Kivity et al. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

[159] Guenter Klas. “Edge Computing and the Role of Cellular Networks”. In: *Computer* 50.10 (2017), pp. 40–49. DOI: 10.1109/MC.2017.3641649.

[160] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896842.

[161] R. Kochendörffer. “Kreyszig, E.: Advanced Engineering Mathematics. J. Wiley & Sons, Inc., New York, London 1962. IX + 856 S. 402 Abb. Preis s. 79.—”. In: *Biometrische Zeitschrift* 7.2 (1965), pp. 129–130. DOI: https://doi.org/10.1002/bimj.19650070232. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/bimj.19650070232. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/bimj.19650070232.

[162] Nane Kratzke. “A brief history of cloud application architectures”. In: *Applied Sciences* 8.8 (Aug. 2018), p. 1368. DOI: 10.3390/app8081368.

[163] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 2017), 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[166] Kin Lane. “Overview of the backend as a service (BaaS) space”. In: *API Evangelist* (2015).

[167] M Lavanya and V Vaithiyanathan. “Load prediction algorithm for dynamic resource allocation”. In: *Indian J Sci Technol* 8 (2015), p. 35.

[168] Michael Le and Yuval Tamir. “ReHype: Enabling VM Survival across Hypervisor Failures”. In: *SIGPLAN Not.* VEE ’11 46.7 (Mar. 2011), 63–74. ISSN: 0362-1340. DOI: 10.1145/2007477.1952692. URL: https://doi.org/10.1145/2007477.1952692.

[169] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. “Evaluation of Production Serverless Computing Environments”. In: (July 2018), pp. 442–450. DOI: 10.13140/RG.2.2.28642.84165. URL: https://www.researchgate.net/publication/324362882.

[171] Ang Li et al. “CloudCmp: Comparing Public Cloud Providers”. In: IMC ’10 (2010), 1–14. DOI: 10.1145/1879141.1879143. URL: https://doi.org/10.1145/1879141.1879143.

[172] Junfeng Li et al. “Understanding Open Source Serverless Platforms: Design Considerations and Performance”. In: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC ’19. Davis, CA, USA: Association for Computing Machinery, 2019, 37–42. ISBN: 9781450370387. DOI: 10.1145/3366623.3368139. URL: https://doi.org/10.1145/3366623.3368139.

[173] Man-Lap Li et al. “Understanding the propagation of hard errors to software and implications for resilient system design”. In: *ASPLOS 2008*. 2008.

[174] Zheng Li et al. “Performance Overhead Comparison between Hypervisor and Container Based Virtualization”. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. 2017, pp. 955–962. DOI: 10.1109/AINA.2017.79.

[175] Ping-Min Lin and Alex Glikson. “Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach”. In: *CoRR* abs/1903.12221 (2019). arXiv: 1903.12221. URL: http://arxiv.org/abs/1903.12221.

[176] Robert F. Ling. “Comparison of Several Algorithms for Computing Sample Means and Variances”. In: *Journal of the American Statistical Association* 69.348 (1974), pp. 859–866. DOI: 10.1080/01621459.1974.10480219. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1974.10480219. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1974.10480219.

[178] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. “Isolation Forest”. In: *2008 Eighth IEEE International Conference on Data Mining*. 2008, pp. 413–422. DOI: 10.1109/ICDM.2008.17.

[179] W. Lloyd et al. “Serverless Computing: An Investigation of Factors Influencing Microservice Performance”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Apr. 2018, pp. 159–169. DOI: 10.1109/IC2E.2018.00039.

[180] Jack Lo. “VMware and CPU virtualization technology”. In: *World Wide Web electronic publication* (2005).

[183] T. Lynn et al. “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Vol. 2017-Decem. IEEE. IEEE Computer Society, Dec. 2017, pp. 162–169. ISBN: 9781538606926. DOI: 10.1109/CloudCom.2017.15.

[184] S. Malla and K. Christensen. “HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)”. In: *Internet Technol. Lett.* 3 (2020).

[185] Sulav Malla and Ken Christensen. “HPC in the cloud: Performance comparison of function as a service (FaaS) vs infrastructure as a service (IaaS)”. In: *Internet Technology Letters* 3.1 (Dec. 2019), e137. DOI: 10.1002/itl2.137.

[188] Johannes Manner et al. “Cold start influencing factors in function as a service”. In: *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*. 2019, pp. 181–188. ISBN: 9781728103594. DOI: 10.1109/UCC-Companion.2018.00054. URL: https://www.jeremydaly.com/15-key-takeaways-from-the-serverless-talk-at-.

[189] G. Mazlami, J. Cito, and P. Leitner. “Extraction of Microservices from Monolithic Software Architectures”. In: *2017 IEEE International Conference on Web Services (ICWS)*. June 2017, pp. 524–531. DOI: 10.1109/ICWS.2017.61.

[190] Garrett McGrath and Paul R. Brenner. “Serverless Computing: Design, Implementation, and Performance”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. IEEE, June 2017, pp. 405–410. DOI: 10.1109/icdcsw.2017.36.

[191] G.J. McLachlan and D. Peel. *Finite Mixture Models*. Wiley Series in Probability and Statistics. Wiley, 2004. ISBN: 9780471654063. URL: https://books.google.de/books?id=c2_fAox0DQoC.

[192] Daniel A Menascé. "Virtualization: Concepts, applications, and performance modeling". In: *Int. CMG Conference*. 2005, pp. 407–414.

[202] Anup Mohan et al. "Agile cold starts for scalable serverless". In: *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. HotCloud'19. USENIX Association. Renton, WA, USA: USENIX Association, 2019, p. 21.

[203] T. S. Mohan. "Migrating into a Cloud". In: *Cloud Computing*. John Wiley & Sons, Ltd, 2011. Chap. 2, pp. 43–56. ISBN: 9780470940105. DOI: https://doi.org/10.1002/9780470940105.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470940105.ch2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470940105.ch2.

[204] S. K. Mohanty, G. Premsankar, and M. di Francesco. "An Evaluation of Open Source Serverless Computing Frameworks". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. Dec. 2018, pp. 115–120. DOI: 10.1109/CloudCom2018.2018.00033. URL: https://doi.org/10.1109/CloudCom2018.2018.00033.

[205] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[206] Gil Neiger et al. "Intel virtualization technology: Hardware support for efficient processor virtualization." In: *Intel Technology Journal* 10.3 (2006).

[210] Jason Nikolai and Yong Wang. "Hypervisor-based cloud intrusion detection system". In: *2014 International Conference on Computing, Networking and Communications (ICNC)* (2014), pp. 989–993.

[211] E. D. Nitto et al. "Supporting the Development and Operation of Multi-cloud Applications: The MODAClouds Approach". In: *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2013, pp. 417–423.

[213] Takahiro Omi, Naonori Ueda, and Kazuyuki Aihara. "Fully Neural Network Based Model for General Temporal Point Processes". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[229] Per-Olov Östberg et al. "Reliable capacity provisioning for distributed cloud/edge/fog computing applications". In: *2017 European conference on networks and communications (EuCNC)*. IEEE. 2017, pp. 1–6.

[230] Manish Parashar et al. "Cloud Paradigms and Practices for Computational and Data-Enabled Science and Engineering". In: *Computing in Science Engineering* 15.4 (2013), pp. 10–18. DOI: 10.1109/MCSE.2013.49.

[231] Maciej Pawlik, Kamil Figiela, and Maciej Malawski. "Performance evaluation of parallel cloud functions". In: *Poster Presented at ICPP* (2018). [Poster Presentation].

[232] Junjie Peng et al. "Modeling for CPU-intensive applications in cloud computing". In: *Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, H* (2015), pp. 20–25. DOI: 10.1109/HPCC-CSS-ICESS.2015.128.

[233] Tobias Pfandzelter and David Bermbach. "tinyFaaS: A lightweight faas platform for edge environments". In: *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE. 2020, pp. 17–24.

[234] Marc-Arhtur Pierre-Louis. "OpenWhisk: A quick tech preview". In: *DeveloperWorks Open, IBM, Feb* 22 (2016), p. 7.

[237] Pooja and A. Pandey. "Impact of memory intensive applications on performance of cloud virtual machine". In: *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*. Mar. 2014, pp. 1–6. DOI: 10.1109/RAECS.2014.6799629.

[242] Ling Qian et al. "Cloud Computing: An Overview". In: *Cloud Computing*. Ed. by Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 626–631. ISBN: 978-3-642-10665-1.

[244] Kunal Rao et al. "ECO: Edge-Cloud Optimization of 5G applications". In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2021, pp. 649–659. DOI: 10.1109/CCGrid51090.2021.00078.

[245] Steven K. Reinhardt and Shubhendu S. Mukherjee. "Transient Fault Detection via Simultaneous Multithreading". In: *SIGARCH Comput. Archit. News*. ISCA '00 28.2 (May 2000), 25–36. ISSN: 0163-5964. DOI: 10.1145/342001.339652. URL: https://doi.org/10.1145/342001.339652.

[246] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. Association for Computing Machinery. New York, NY, USA: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391236. URL: https://doi.org/10.1145/2391229.2391236.

[247] Xiaona Ren, Rongheng Lin, and Hua Zou. "A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast". In: *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*. IEEE, Sept. 2011. DOI: 10.1109/ccis.2011.6045063.

[248] Andre Oliver Richter. "Mitigating and Resolving Performance Isolation Issues of PCIe Passthrough and SR-IOV in Multi-Core Virtualization". Dissertation. München: Technische Universität München, 2017.

[250] Fernando Rodríguez-Haro et al. "A summary of virtualization techniques". In: *Procedia Technology* 3 (2012). The 2012 Iberoamerican Conference on Electronics Engineering and Computer Science, pp. 267–272. ISSN: 2212-0173. DOI: https://doi.org/10.1016/j.protcy.2012.03.029. URL: https://www.sciencedirect.com/science/article/pii/S2212017312002587.

[253] Gor Safaryan et al. "SLAM: SLO-Aware Memory Optimization for Serverless Applications". In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022, pp. 30–39. DOI: 10.1109/CLOUD55607.2022.00019.

[254] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath. "Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues". In: *2010 Second International Conference on Computer and Network Technology*. 2010, pp. 222–226. DOI: 10.1109/ICCNT.2010.49.

[255] M. Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. DOI: 10.1109/mprv.2009.82.

[256] Mahadev Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[257] Mahadev Satyanarayanan et al. "The case for vm-based cloudlets in mobile computing". In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23.

[258] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. “OpenStack: toward an open-source solution for cloud computing”. In: *International Journal of Computer Applications* 55.3 (2012), pp. 38–42.

[263] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. “Architectural implications of function-as-a-service computing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 1063–1075.

[264] Yizhou Shan et al. “LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, 69–87. ISBN: 9781931971478.

[265] Vaishaal Shankar et al. “Numpywren: Serverless linear algebra”. In: *arXiv preprint arXiv:1810.09679*. SoCC ’20 (2018), 281–295. DOI: 10.1145/3419111.3421287. URL: https://doi.org/10.1145/3419111.3421287.

[266] G.Siva Shanmugam and N.Ch.S. N. Iyengar. “Effort of Load Balancer to Achieve Green Cloud Computing: A Review”. In: *International Journal of Multimedia and Ubiquitous Engineering* 11.3 (Mar. 2016), pp. 317–332. DOI: 10.14257/ijmue.2016.11.3.30.

[269] Oleksandr Shchur, Marin Biloš, and Stephan Günnemann. “Intensity-Free Learning of Temporal Point Processes”. In: *CoRR* abs/1909.12127 (2019). arXiv: 1909.12127. URL: http://arxiv.org/abs/1909.12127.

[270] Oleksandr Shchur et al. “Neural Temporal Point Processes: A Review”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*. Ed. by Zhi-Hua Zhou. Survey Track. International Joint Conferences on Artificial Intelligence Organization, Aug. 2021, pp. 4585–4593. DOI: 10.24963/ijcai.2021/623. URL: https://doi.org/10.24963/ijcai.2021/623.

[271] Jiacheng Shen et al. “Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms”. In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 2021, pp. 194–204. DOI: 10.1109/ICDCS51616.2021.00027.

[272] Olena Skarlat et al. “Resource provisioning for IoT services in the fog”. In: *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*. IEEE. 2016, pp. 32–39.

[273] Christopher Peter Smith et al. “FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters”. In: *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. 2022, pp. 17–25. DOI: 10.1109/ICFEC54809.2022.00010.

[274] Gaurav Somani and Sanjay Chaudhary. “Application Performance Isolation in Virtualization”. In: *2009 IEEE International Conference on Cloud Computing*. 2009, pp. 41–48. DOI: 10.1109/CLOUD.2009.78.

[275] Vladimir Sor and Satish Narayana Srirama. “A Statistical Approach for Identifying Memory Leaks in Cloud Applications”. In: *CLOSER*. 2011.

[276] Josef Spillner. “Resource Management for Cloud Functions with Memory Tracing, Profiling and Autotuning”. In: *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*. New York, NY, USA: ACM, Dec. 2020, pp. 13–18. ISBN: 9781450382045. DOI: 10.1145/3429880.3430094. URL: https://doi.org/10.1145/3429880.3430094.

[277] Josef Spillner, Cristian Mateos, and David A. Monge. “FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC”. In: *High Performance Computing*. Ed. by Esteban Mocskos and Sergio Nesmachnow. Springer International Publishing. Cham: Springer International Publishing, 2018, pp. 154–168. ISBN: 978-3-319-73353-1.

[278] Stefan Stasiewicz. “Worth Getting Hyped Up Over Hyper-V?” In: *21st Annual Conference of NACCQ*. 2008.

[279] Markus Steinbach et al. “TppFaaS: Modeling Serverless Functions Invocations via Temporal Point Processes”. In: *IEEE Access* 10 (2022), pp. 9059–9084. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3144078.

[281] Kun Suo et al. “Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing”. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 2021, pp. 433–443. DOI: 10.1109/Cluster48925.2021.00018.

[282] Amoghavarsha Suresh and Anshul Gandhi. “FnSched: An Efficient Scheduler for Serverless Functions”. In: Dec. 2019, pp. 19–24. ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368136.

[283] Jun Suzuki et al. “Multi-root share of single-root I/O virtualization (SR-IOV) compliant PCI Express device”. In: *2010 18th IEEE Symposium on High Performance Interconnects*. IEEE. 2010, pp. 25–31.

[284] Sharvari T and Sowmya Nag K. “A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming”. In: *CoRR* abs/1912.03715 (2019). arXiv: 1912.03715. URL: http://arxiv.org/abs/1912.03715.

[285] Davide Taibi, Josef Spillner, and Konrad Wawruch. “Serverless Computing-Where Are We Now, and Where Are We Heading?” In: *IEEE Software* 38.1 (2021), pp. 25–31. ISSN: 19374194. DOI: 10.1109/MS.2020.3028708.

[286] Tarik Taleb et al. “On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration”. In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1657–1681. DOI: 10.1109/COMST.2017.2705720.

[292] Liang Tong, Yong Li, and Wei Gao. “A hierarchical edge cloud architecture for mobile computing”. In: *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE. 2016, pp. 1–9.

[293] Ruixia Tong and Xiongfeng Zhu. “A Load Balancing Strategy Based on the Combination of Static and Dynamic”. In: *2010 2nd International Workshop on Database Technology and Applications*. IEEE, Nov. 2010. DOI: 10.1109/dbta.2010.5658951.

[295] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[298] Tim Verbelen et al. “Cloudlets: Bringing the Cloud to the Mobile User”. In: *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*. MCS ’12. Low Wood Bay, Lake District, UK: Association for Computing Machinery, 2012, 29–36. ISBN: 9781450313193. DOI: 10.1145/2307849.2307858. URL: https://doi.org/10.1145/2307849.2307858.

[299] Massimo Villari et al. “Osmotic computing: A new paradigm for edge/cloud integration”. In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83.

[302] Liang Wang et al. “Peeking behind the curtains of serverless platforms”. In: *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*. USENIX Association. 2018, pp. 133–146. ISBN: 978-1-939133-02-1. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

[303] Ben Weissman and Anthony E. Nocentino. “Azure Arc-Enabled Data Services”. In: *Azure Arc-Enabled Data Services Revealed: Early First Edition Based on Public Preview*. Berkeley, CA: Apress, 2021, pp. 25–50. ISBN: 978-1-4842-6705-9. DOI: 10.1007/978-1-4842-6705-9_2. URL: https://doi.org/10.1007/978-1-4842-6705-9_2.

[304] CNCF Serverless WG. *Cncf wg-serverless whitepaper v1. 0*. https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf. [Online; Accessed: 15-July-2020]. March 2018. URL: https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf.

[309] Jonathan Newcomb Swirsky Whitney et al. *Hybrid cloud infrastructures*. US Patent 9,122,552. Sept. 2015.

[311] Philipp A. Witte et al. “Serverless seismic imaging in the cloud”. In: *CoRR* abs/1911.12447 (2019). arXiv: 1911.12447. URL: http://arxiv.org/abs/1911.12447.

[313] Yichen Xie and Alex Aiken. “Context- and Path-sensitive Memory Leak Detection”. In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 115–125. ISSN: 0163-5948. DOI: 10.1145/1095430.1081728. URL: http://doi.acm.org/10.1145/1095430.1081728.

[314] Ying Xiong et al. “Extend cloud to edge with KubeEdge”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 373–377.

[315] Xin Xu, Ron C. Chiang, and H. Howie Huang. “Xentry: Hypervisor-Level Soft Error Detection”. In: *2014 43rd International Conference on Parallel Processing*. 2014, pp. 341–350. DOI: 10.1109/ICPP.2014.43.

[316] Sanjay Yadav and Sanyam Shukla. “Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification”. In: *2016 IEEE 6th International conference on advanced computing (IACC)*. IEEE. 2016, pp. 78–83.

[317] Ethan G Young et al. “The True Cost of Containing: A {gVisor} Case Study”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019.

[318] Yang Yu. *Os-level virtualization and its applications*. State University of New York at Stony Brook, 2007.

[319] Yang Zhang, Nirvana Meratnia, and Paul J.M. Havinga. “Distributed online outlier detection in wireless sensor networks using ellipsoidal support vector machine”. In: *Ad Hoc Networks* 11.3 (2013), pp. 1062–1074. ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2012.11.001. URL: http://www.sciencedirect.com/science/article/pii/S1570870512002041.

[321] Tetiana Zubko et al. “MAFF: Self-adaptive Memory Optimization for Serverless Functions”. In: *Service-Oriented and Cloud Computing*. Ed. by Fabrizio Montesi, George Angelos Papadopoulos, and Wolf Zimmermann. Cham: Springer International Publishing, 2022, pp. 137–154. ISBN: 978-3-031-04718-3.

Webliography

[1] *3 ways to optimize Cloud Run response times*. Accessed: 2022-08-27. 2020. URL: https://cloud.google.com/blog/topics/developers-practitioners/3-ways-optimize-cloud-run-response-times.

[2] *About Ansible*. 2020. URL: https://docs.ansible.com/ansible/latest/index.html#about-ansible (visited on 07/09/2020).

[3] *About Playbooks*. 2020. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#playbooks-intro (visited on 07/09/2020).

[4] *About VirtualBox*. Accessed: 2022-06-20. 2021. URL: https://www.virtualbox.org/wiki/VirtualBox.

[10] Amazon Lambda. *AWS Lambda*. Accessed on 09/24/2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html (visited on 03/09/2021).

[11] *An introduction to Azure Functions*. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[13] Apache. *Documentation*. 2020. URL: https://openwhisk.apache.org/documentation.html (visited on 07/20/2020).

[14] Apache. *OpenWhisk CLI*. 2017. URL: https://github.com/apache/openwhisk/blob/master/docs/cli.md#openwhisk-cli (visited on 07/11/2020).

[16] Architect. *Project philosophy*. [Online; Accessed: 4-Feburary-2020]. 2020. URL: https://arc.codes/intro/philosophy.

[19] *AT&T and Google Cloud forge 5G edge compute partnership for enterprises*. Accessed: 2022-08-27. 2020. URL: https://www.fiercetelecom.com/telecom/at-t-and-google-cloud-forge-5g-edge-compute-partnership-for-enterprises/.

[20] *AT&T to run its mobility network on Microsoft's Azure for Operators cloud, delivering cost-efficient 5G services at scale*. Accessed: 2022-08-27. 2021. URL: https://news.microsoft.com/2021/06/30/att-to-run-its-mobility-network-on-microsofts-azure-for-operators-cloud-delivering-cost-efficient-5g-services-at-scale/.

[21] AWS. *Application Load Balancers as targets*. 2020. URL: https://docs.aws.amazon.com/elasticload\-balancing/latest/network/application-load-balancer-target.html (visited on 04/17/2021).

[22] AWS. *AWS Elastic LoadBalancer Limits*. Accessed: 2021-01-10. URL: $https://docs.aws.amazon.com/de_de/elasticloadbalancing/latest/userguide/how-elastic-load-balancing-works.html$.

[23] AWS. *AWS Fargate*. 2019. URL: https://github.com/virtual-kubelet/aws-fargate (visited on 10/01/2021).

[24] AWS. *AWS SDK for Python (Boto3) Documentation*. [Online; Accessed: 14-Feburary-2020]. URL: https://docs.aws.amazon.com/pythonsdk/.

[25] AWS. *How do I attach an Elastic IP address to new or existing internet-facing Network Load Balancers?* 2022. URL: https://aws.amazon.com/premiumsupport/knowledge-center/elb-attach-elastic-ip-to-public-nlb/ (visited on 07/30/2022).

[26] AWS. *I need a static IP address for my Application Load Balancer. How can I register an Application Load Balancer behind a Network Load Balancer?* 2022. URL: https://aws.amazon.com/premiumsupport/knowledge-center/alb-static-ip/ (visited on 07/30/2022).

[27] AWS. *Lambda functions as targets*. 2020. URL: https://docs.aws.amazon.com/elasticload\-balancing/latest/application/lambda-functions.html (visited on 04/17/2021).

[28] AWS. *Lambda runtimes - AWS Lambda*. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (visited on 06/16/2021).

[29] AWS. *Package google.monitoring.v3*. [Online; Accessed: 14-Feburary-2020]. URL: https://cloud.google.com/monitoring/api/ref_v3/rpc/google.monitoring.v3.

[30] AWS. *Serverless Architectures with AWS Lambda*. 2017. URL: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf (visited on 06/16/2021).

[31] AWS. *What is a Network Load Balancer?* 2020. URL: https://docs.aws.amazon.com/elasticload\-balancing/latest/network/introduction.html (visited on 04/17/2021).

[32] AWS. *What is an Application Load Balancer?* 2020. URL: https://docs.aws.amazon.com/\-elasticloadbalancing/latest/application/introduction.html (visited on 04/17/2021).

[33] AWS. *What Is AWS X-Ray?* [Online; Accessed: 4-Feburary-2020]. 2020. URL: https://docs.aws.amazon.com/xray/latest/devguide/aws-xray.html.

[34] *AWS Compute Optimizer*. (Accessed on 06/17/2021). 2021. URL: https://aws.amazon.com/compute-optimizer/.

[35] AWS Elastic Load Balancing. Accessed 09/24/2020. URL: https://aws.amazon.com/elastic\-loadbalancing/.

[36] *AWS IoT Greengrass - Amazon Web Services*. (Accessed on 07/27/2020). URL: https://aws.amazon.com/greengrass/.

[37] *AWS Lambda*. URL: https://aws.amazon.com/lambda/.

[38] *AWS Lambda Pricing*. 2020. URL: https://aws.amazon.com/lambda/pricing/ (visited on 05/31/2020).

[39] *AWS Lambda releases*. 2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html (visited on 05/31/2020).

[40] *AWS Lambda Shim*. Accessed: 2022-08-20. 2022. URL: https://github.com/ffleet/shim/.

[41] *AWS Lambda – Pricing*. (Accessed on 07/30/2020). URL: https://aws.amazon.com/lambda/pricing/.

[42] *AWS Local Zones*. Accessed: 2022-08-27. 2021. URL: https://aws.amazon.com/about-aws/global-infrastructure/localzones/.

[43] *AWS Nitro System*. Accessed: 2022-08-27. 2022. URL: https://aws.amazon.com/ec2/nitro/.

[44] Azure. *Azure Functions hosting options*. Accessed on 02/18/2021. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale (visited on 03/09/2021).

[45] Azure. *Kubernetes Virtual Kubelet with ACI*. 2022. URL: https://github.com/virtual-kubelet/azure-aci (visited on 02/01/2022).

[54] Rajesh Bhojwani. *Design Patterns for Microservice-To-Microservice Communication - DZone Microservices*. Dec. 2018. URL: https://dzone.com/articles/design-patterns-for-microservice-communication.

[59] Apache Brooklyn. *The Theory behind Brooklyn*. Accessed: 2020-01-10. URL: https://brooklyn.apache.org/learnmore/theory.html.

[63] Renato Byrro. *Can We Solve Serverless Cold Starts?* Accessed: 2020-04-17. 2019. URL: https://dashbird.io/blog/can-we-solve-serverless-cold-starts/.

[64] cadvisor. *Monitoring cAdvisor with Prometheus*. 2019. URL: https://github.com/google/cadvisor/blob/master/docs/storage/prometheus.md (visited on 06/11/2019).

[68] Alex Casalboni. *Announcing AWS Lambda Function URLs: Built-in HTTPS Endpoints for Single-Function Microservices*. 2022. URL: https://aws.amazon.com/blogs/aws/announcing-aws-lambda-function-urls-built-in-https-endpoints-for-single-function-microservices/ (visited on 04/29/2022).

[69] Alex Casalboni. *AWS Lambda Power Tuning*. URL: https://github.com/alexcasalboni/aws-lambda-power-tuning (visited on 03/09/2021).

[78] *Citrix Hypervisor Technical overview*. Accessed: 2022-06-20. 2022. URL: https://docs.citrix.com/en-us/citrix-hypervisor/8-2/technical-overview.html.

[79] *Cloud Functions Overview*. (Accessed on 08/22/2020). URL: https://cloud.google.com/functions/docs/concepts/overview.

[80] CloudFlare. *Why use serverless computing?* Accessed: 2020/12/16. URL: https://www.cloudflare.com/learning/serverless/why-use-serverless/.

[81] CloudWatch. *Amazon CloudWatch Documentation*. URL: https://docs.aws.amazon.com/cloudwatch/index.html (visited on 06/23/2021).

[82] Controlling Scaling Behavior. Accessed 09/24/2020. URL: https://cloud.google.com/functions/docs/max-instances.

[85] Michael Crosby. *What is containerd?* Accessed on 17.06.2019 18:57. Docker Inc. 2017. URL: https://blog.docker.com/2017/08/what-is-containerd-runtime/.

[89] *Description of Windows Virtual PC*. Accessed: 2022-06-20. 2021. URL: https://support.microsoft.com/en-us/topic/description-of-windows-virtual-pc-262c8961-90e5-1125-654f-d87cd5ba16f8.

[94] *Dynamic Configuration with the HAProxy Runtime API*. Accessed: 2021-10-10. 2017.

[100] Alex Ellis. *A bright 2019 for OpenFaaS*. 2019. URL: https://blog.alexellis.io/openfaas-bright-2019/ (visited on 07/11/2020).

[108] Martin Fowler. *Circuit Breaker*. Accessed: 2021-01-10. 2014. URL: https://martinfowler.com/bliki/CircuitBreaker.html.

[113] *gcloud CLI overview*. Accessed: 2022-08-27. 2020. URL: https://cloud.google.com/sdk/gcloud.

[116] Google. *Cloud Functions*. [Accessed: 4 September 2021]. URL: https://cloud.google.com/functions.

[117] Google Cloud. *Select metrics when using Metrics Explorer*. URL: https://cloud.google.com/monitoring/charts/metrics-selector (visited on 03/09/2021).

[118] *Google Cloud Recommendations*. (Accessed on 06/17/2021). 2018. URL: https://cloud.google.com/compute/docs/instances/apply-machine-type-recommendations-for-instances.

[119] GoogleCloud. *Cloud Functions Execution Environment*. (Accessed on 08/22/2020). URL: https://cloud.google.com/functions/docs/concepts/exec.

[120] GoogleCloud. *Cloud Functions Pricing*. (Accessed on 08/22/2020). URL: https://cloud.google.com/functions/pricing.

[121] Grafana. *Grafana Dashboards*. Accessed: 2021-05-27. 2019. URL: https://grafana.com/docs/ (visited on 06/11/2019).

[125] *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer*. Accessed: 2021-10-10. 2021.

[126] *HAProxy - They use it !* Accessed: 2021-10-10. 2021.

[130] Andy Honig and Nelly Porter. *7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext*. Accessed: 2022-06-20. 2017. URL: https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext.

[132] *How Terraform Works - CLI*. 2020. URL: https://www.terraform.io/ (visited on 07/07/2020).

[133] *Hypervisor security on the Azure fleet*. Accessed: 2022-08-27. 2022. URL: https://docs.microsoft.com/en-us/azure/security/fundamentals/hypervisor.

[134] IBM. *IBM Cloud Functions*. [Accessed: 7 September 2021]. URL: https://cloud.ibm.com/functions/ (visited on 08/01/2021).

[135] Influxdata. *Influxdb*. Accessed: 2021-05-27. URL: https://www.influxdata.com/products/influxdb/.

[136] *Introduction to HAProxy ACLs*. Accessed: 2021-10-10. 2018.

[137] *Introduction to Hyper-V on Windows 10*. Accessed: 2022-06-20. 2022. URL: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/.

[141] Frazer Jamieson. *Losing the server?* 2017. URL: https://www.bcs.org/content-hub/losing-the-server/ (visited on 07/11/2020).

[144] Anshul Jindal. *FDN Virtual Kubelet*. 2022. URL: https://github.com/Function-Delivery-Network/virtual-kubelet (visited on 04/01/2022).

[145] Anshul Jindal. *Multi-Serverless-Deployment*. 2021. URL: https://github.com/ansjin/multi-cloud-serverless-deployment (visited on 07/11/2021).

[164] virtual kubelet. *Virtual Kubelet*. 2021. URL: https://github.com/virtual-kubelet/virtual-kubelet (visited on 09/01/2021).

[165] *Lambda function URLs*. Accessed: 2022-08-27. 2022. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-urls.html.

[170] *Leibniz Supercomputing Centre*. (Accessed on 07/28/2020). URL: https://www.lrz.de/english/.

[177] Linux. *dd(1) - Linux manual page*. URL: https://man7.org/linux/man-pages/man1/dd.1.html (visited on 07/14/2021).

[181] *Load Balancing and Reverse Proxying with Nginx, Updated*. Accessed: 2021-01-12. 2013. URL: https://spin.atomicobject.com/2013/07/08/nginx-load-balancing-reverse-proxy-updated/.

[182] *LRZ: Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften*. (Accessed on 07/30/2020). URL: https://www.lrz.de/.

[186] Managing concurrency for a Lambda function. Accessed 09/24/2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html.

[187] *Managing Lambda provisioned concurrency*. Accessed: 2022-08-27. 2022. URL: https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.

[193] Microsoft. *Azure Durable Functions*. [Accessed: 4 September 2021]. URL: https://docs.microsoft.com/en-us/azure/azure-functions/durable/.

[194] Microsoft. *Azure Functions*. [Accessed: 4 September 2021]. URL: https://azure.microsoft.com/de-de/services/functions/.

[195] MinIO. *MinIO Object Storage*. Accessed: 2021-05-27. URL: https://min.io/product/overview.

[196] MinIO, Inc. *Bucket Replication*. URL: https://docs.min.io/minio/baremetal/replication/replication-overview.html (visited on 05/10/2021).

[197] MinIO, Inc. *mc mirror*. URL: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc/mc-mirror.html#command-mc-mirror (visited on 05/10/2021).

[198] MinIO, Inc. *MinIO - Go Client API Reference*. URL: https://docs.min.io/docs/golang-client-api-reference.html (visited on 05/10/2021).

[199] MinIO, Inc. *MinIO - JavaScript Client API Reference*. URL: https://docs.min.io/docs/javascript-client-api-reference.html (visited on 05/10/2021).

[200] MinIO, Inc. *MinIO Client (mc)*. URL: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc.html (visited on 05/10/2021).

[201] *MinIO Quickstart Guide*. (Accessed on 07/28/2020). URL: https://docs.min.io/docs/minio-quickstart-guide.html.

[207] NGINX. *NGINX Load Balancing Algorithms*. Accessed: 2020-01-10. URL: https://www.nginx.com/blog/choosing-nginx-plus-load-balancing-techniques/.

[208] Nginx. *Nginx: The High-Performance Web Server and Reverse Proxy*. Accessed 09/24/2020. Sept. 2008. URL: https://www.nginx.com/.

[209] NGINX. *What Is DNS Load Balancing?* Accessed: 2021-01-12. URL: https://www.nginx.com/resources/glossary/dns-load-balancing/.

[212] Ekaterina Novoseltseva. *Benefits of Microservices Architecture Implementation*. [Online; Accessed: 23-March-2020]. 2017. URL: https://dzone.com/articles/benefits-amp-examples-of-microservices-architectur.

[214] OpenfaaS. *faas-idler: Scale OpenFaaS functions to zero replicas after a period of inactivity*. 2018. URL: https://github.com/openfaas-incubator/faas-idler (visited on 07/11/2020).

[215] OpenfaaS. *faas-netes*. 2017. URL: https://github.com/openfaas/faas-netes (visited on 07/11/2020).

[216] OpenfaaS. *Kubernetes HPAv2 with OpenFaaS*. 2019. URL: https://docs.openfaas.com/tutorials/kubernetes-hpa/ (visited on 07/11/2020).

[217] OpenfaaS. *OpenFaaS stack*. 2019. URL: https://docs.openfaas.com/architecture/stack/ (visited on 07/11/2020).

[218] *OpenFaaS watchdog*. 2016. URL: https://docs.openfaas.com/architecture/watchdog/ (visited on 05/31/2020).

[219] Openstack. *OpenStack Zun*. 2020. URL: https://github.com/virtual-kubelet/openstack-zun (visited on 04/01/2022).

[220] OpenTelemetry. *OpenTelemetry Collector: Batch Processor*. [Accessed: 30 April 2021]. URL: https://github.com/open-telemetry/opentelemetry-collector/tree/main/processor/batchprocessor.

[221] OpenTelemetry. *OpenTelemetry Collector: OLTP Receiver*. [Accessed: 30 April 2021]. URL: https://github.com/open-telemetry/opentelemetry-collector/tree/main/receiver/otlpreceiver.

[222] OpenTelemetry. *OpenTelemetry Javascript*. [Accessed: 5 May 2021]. URL: https://opentelemetry.io/docs/js/.

[223] OpenWhisk. *Apache OpenWhisk Composer*. [Accessed: 4 September 2021]. URL: https://github.com/apache/openwhisk-composer.

[224] OpenWhisk. *OpenWhisk Annotations*. [Accessed: 30 April 2021]. URL: https://github.com/apache/openwhisk/blob/master/docs/annotations.md.

[225] OpenWhisk. *OpenWhisk: Open Source Serverless Cloud Platform*. [Accessed: 16 May 2021]. URL: https://openwhisk.apache.org.

[226] Apache OpenWhisk. *Apache openwhisk is a serverless, open source cloud platform*. [Online; Accessed: 4-Feburary-2020]. 2018. URL: https://openwhisk.apache.org/documentation.html.

[227] *Oracle VM Server for x86 Virtualization and Management*. Accessed: 2022-06-20. 2021. URL: https://www.oracle.com/uk/a/ocom/docs/ovm-server-for-x86-459312.pdf.

[228] Antoni Orfin. *How Droplr Scales to Millions With The Serverless Framework*. [Online; Accessed: 14-Feburary-2020]. 2017. URL: https://www.serverless.com/blog/how-droplr-scales-to-millions-serverless-framework.

[235] Google Cloud Platform. *Cloud Load Balancing*. 2022. URL: https://cloud.google.com/load-balancing (visited on 04/30/2022).

[236] Google Cloud Platform. *Serverless network endpoint groups overview*. Accessed: 2021-05-27. URL: https://cloud.google.com/load-balancing/docs/negs/serverless-neg-concepts.

[238] *Power Measurement Framework for edge-enabled IoT devices*. Accessed: 2022-08-20. 2022. URL: https://github.com/Manu10744/esp32-edge-energy-measurement/.

[239] The Linux Foundation Prometheus Authors. *Prometheus - Monitoring system & time-series database*. 2019. URL: https://prometheus.io/docs/introduction/overview/ (visited on 06/11/2019).

[240] The Linux Foundation Prometheus Autors. *MONITORING LINUX HOST METRICS WITH THE NODE EXPORTER*. 2019. URL: https://prometheus.io/docs/guides/node-exporter/ (visited on 06/11/2019).

[241] PyData. *pandas.DataFrame*. [Online; Accessed: 19-June-2021]. URL: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html.

[243] *rancher/k3s: Lightweight Kubernetes*. (Accessed on 07/28/2020). URL: https://github.com/rancher/k3s.

[249] Mike Roberts. *Serverless Architectures*. Accessed: 2020-04-17. 2018. URL: https://martinfowler.com/articles/serverless.html.

[251] *Roles*. 2020. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html#roles (visited on 07/09/2020).

[252] *runc*. Accessed: 2022-08-27. 2020. URL: https://github.com/opencontainers/runc.

[259] Serverless. *Documentation*. [Online; Accessed: 4-Feburary-2020]. 2020. URL: https://serverless.com/framework/docs/.

[260] Amazon Web Services. *AWS Lambda*. [Accessed: 4 September 2021]. URL: https://aws.amazon.com/lambda/.

[261] Amazon Web Services. *AWS Step Functions*. [Accessed: 4 September 2021]. URL: https://aws.amazon.com/step-functions/.

[262] Amazon Web Services. *AWS Step Functions Features*. [Accessed: 4 September 2021]. URL: https://aws.amazon.com/step-functions/features/.

[267] Oleksandr Shchur. *Loss with NLL of mark and MAE of inter-event time*. [Accessed: 13 September 2021]. 2021. URL: https://github.com/shchur/ifl-tpp/issues/14.

[268] Oleksandr Shchur. *Temporal Point Processes 1: The Conditional Intensity Function*. [Accessed: 16 June 2021]. 2020. URL: https://shchur.github.io/blog/2020/tpp1-conditional-intensity/.

[280] Ion Stoica and Devin Petersohn. *Two missing links in Serverless Computing: Stateful Computation and Placement Control*. 2019. URL: https://medium.com/riselab/two-missing-links-in-serverless-computing-stateful-computation-and-placement-control-964c3236d18 (visited on 04/17/2021).

[287] *The Four Essential Sections of an HAProxy Configuration*. Accessed: 2021-10-10. 2018.

[288] *The performance of modern AI for millions of devices NVIDIA Jetson Nano*. (Accessed on 07/28/2020). URL: https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-nano/.

[289] The PostgreSQL Global Development Group. *PostgreSQL - Documentation*. URL: https://www.postgresql.org/docs/ (visited on 04/23/2021).

[290] *The regression score*. Accessed on 12/17/2020. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html.

[291] Markus Thömmes. *Squeezing the milliseconds: How to make serverless platforms blazing fast!* [Online; Accessed: 14-Feburary-2020]. 2017. URL: https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0.

[294] Andrew Tridgell and Paul Mackerras. *rsync(1) man page*. URL: https://download.samba.org/pub/rsync/rsync.1 (visited on 05/10/2021).

[296] *Use Image streaming to pull container images*. Accessed: 2022-08-27. 2022. URL: https://cloud.google.com/kubernetes-engine/docs/how-to/image-streaming.

[297] *Using AWS Lambda with CloudFront Lambda@Edge*. Accessed: 2020-06-20. 2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html.

[300] *VMware ESXi*. Accessed: 2022-06-20. 2021. URL: https://www.vmware.com/content/vmware/vmware-published-sites/us/products/esxi-and-esx.html.html.

[301] *VMware Workstation Pro Documentation*. Accessed: 2022-06-20. 2021. URL: https://docs.vmware.com/en/VMware-Workstation-Pro/index.html.

[305] *What is Azure private multi-access edge compute?* Accessed: 2022-08-27. 2022. URL: https://docs.microsoft.com/en-us/azure/private-multi-access-edge-compute-mec/overview.

[306] *What is k6?* (Accessed on 07/28/2020). URL: https://k6.io/docs/.

[307] *What is rkt?* Accessed: 2022-08-27. 2021. URL: https://www.redhat.com/en/topics/containers/what-is-rkt.

[308] *What's LXC?* Accessed: 2022-08-27. 2021. URL: https://linuxcontainers.org/lxc/introduction/.

[310] *Wikipedia Traffic Data Exploration*. Accessed: 2021-10-05. 2021. URL: https://www.kaggle.com/muonneutrino/wikipedia-traffic-data-exploration/.

[312] *XenCenter*. Accessed: 2022-08-27. 2022. URL: https://docs.citrix.com/en-us/xencenter.html.

[320] Zipkin. *Zipkin*. [Accessed: 4 September 2021]. URL: https://zipkin.io/.

[322] Emrah Şamdan. *Dealing with cold starts in AWS Lambda*. [Online; Accessed: 14-Feburary-2020]. 2018. URL: https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532.