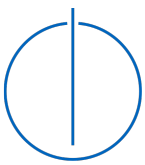
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Framework for Performance Regression Evaluation of Serverless Applications

Hady Yasser Dossoki Mohamed



DEPARTMENT OF INFORMATICS

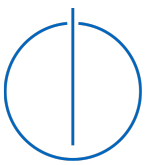
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Framework for Performance Regression Evaluation of Serverless Applications

Umgebung zur Erkennung von Performance-Regressionen bei Serverless Anwendungen

Author:	Hady Yasser Dossoki Mohamed
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	M.Sc. Anshul Jindal
Submission Date:	15.07.2022



Acknowledgments

I want to start by thanking my supervisor Anshul Jindal, who allowed me to work on such an exciting topic. Thank you for everything you have taught me throughout this thesis and the constant helpful feedback. It has been a pleasure working with you.

I would also like to thank Prof. Dr. Gerndt, who introduced me to cloud computing through his lectures, which turned out to be the field I am currently working in. Thank you for everything.

Furthermore, I would like to thank my colleagues and friends, especially Mohamed El Zarei. In addition, I would also like to thank my family for their constant support throughout this journey.

Finally, I would like to thank my partner, Rania, for inspiring and pushing me through this journey. I would not have been able to make it without your constant support.

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.07.2022

Hady Yasser Dossoki Mohamed

Abstract

Recent developments in cloud computing have allowed for the rise of Function-as-a-Service (FaaS), which has been increasing in popularity. This popularity results from its various features in terms of scalability and removing the burden of maintaining the underlying infrastructure. Despite its increasing popularity, the performance of Function-as-a-Service (FaaS) based application over a prolonged period remains understudied. With the increased adoption of FaaS, such performance insights become essential. FaaS based applications also introduce an increased overhead for debugging performance issues due to their sparse nature and the lack of control over the underlying infrastructure. In this thesis, we present **RegX**: An implementation of an automatic, serverless regression testing framework. This framework measures different metrics after every change introduced to a serverless application. Throughout this thesis, we showcase how **RegX** can provide additional performance insights and help detect performance issues as they are introduced. Using **RegX**, we track the performance of two applications over different commits. The results we collected from **RegX** show that the first application, where the developers actively made an effort to improve the architecture at the expense of more resources, results in improved performance. On the other hand, the other application witnessed a degradation in performance and an increase in resource usage. This suggests that performance is dependent on developer choices. Using a tool such as **RegX** can help identify specific code changes that result in performance degradation.

Abstrakt

Die neuesten Entwicklungen im Cloud-Computing haben den Aufstieg von Function-as-a-Service (FaaS) ermöglicht, das immer beliebter wird. Diese Popularität ergibt sich aus seinen verschiedenen Merkmalen in Bezug auf Skalierbarkeit und Beseitigung der Belastung durch die Wartung der zugrunde liegenden Infrastruktur. Trotz ihrer zunehmenden Popularität bleibt die Leistung von FaaS-basierten Anwendungen über einen längeren Zeitraum zu wenig untersucht. Mit der zunehmenden Einführung von FaaS werden solche Einblicke in die Leistung unerlässlich. FaaS-basierte Anwendungen führen aufgrund ihrer spärlichen Natur und der fehlenden Kontrolle über die zugrunde liegende Infrastruktur auch zu einem erhöhten Overhead für das Debuggen von Leistungsproblemen. In dieser Arbeit stellen wir **RegX** vor: Eine Implementierung eines automatischen, serverlosen Regressionstest-Frameworks. Dieses Framework misst verschiedene Metriken nach jeder Änderung, die an einer serverlosen Anwendung vorgenommen wird. In dieser Arbeit zeigen wir, wie **RegX** zusätzliche Einblicke in die Leistung liefern und dabei helfen kann, Leistungsprobleme zu erkennen, sobald sie auftreten. Mit **RegX** verfolgen wir die Leistung von zwei Anwendungen über verschiedene Commits hinweg. Die Ergebnisse, die wir von **RegX** gesammelt haben, zeigen, dass die erste Anwendung, bei der sich die Entwickler aktiv bemüht haben, die Architektur auf Kosten von mehr Ressourcen zu verbessern, zu einer verbesserten Leistung führt. Auf der anderen Seite erlebte die andere Anwendung eine Verschlechterung der Leistung und eine Zunahme der Ressourcennutzung. Dies deutet darauf hin, dass die Leistung von den Entscheidungen des Entwicklers abhängt. Die Verwendung eines Tools wie **RegX** kann dabei helfen, bestimmte Codeänderungen zu identifizieren, die zu LeistungseinbuSSen führen.

Contents

Acronyms

AWS Amazon Web Services. 5–8, 25, 28, 56

CDK Cloud Development Kit. 12, 23

CQRS Command and Query Responsibility Segregation. 34

FaaS Function-as-a-Service. iv, v, 1–3, 5, 6, 8, 9, 16–18, 24, 25, 30, 56

FDN Function Delivery Network. 30

IAC Infrastructure As Code. 3, 11, 12

IAM Identity and Access Management. 8, 22, 23

PaaS Platform-as-a-Service. 5

PPTAM Production and performance application monitor. 16

S3 Simple Storage Service. 6, 7, 20, 22–24

SAAF Serverless Application Analytics Framework. 17

SNS Simple Notification service. 6, 7, 34

VM Virtual Machines. 5

VUs Virtual Users. 30

1. Introduction

Serverless computing is an execution model in which a cloud service provider dynamically maintains the server's computational resources [43]. An example is Function-as-a-Service (FaaS) - A form of serverless computing that allows the execution of pieces of code encapsulated within a function to be executed upon function invocation [3]. FaaS and serverless architectures, in general, have increased in popularity over the past few years. This increase in popularity is due to the fine granularity over billing and scalability provided by most providers (e.g., Amazon Web Services and Google Cloud) [43]. They also remove the burden of maintaining the infrastructure, as this responsibility shifts to the provider.

Since FaaS introduces different ways of structuring and implementing applications, some decisions become subjective and can be influenced by the developer's knowledge. This can result in a suboptimal performance that underperforms after introducing a change. The result may be a symptom of the golden hammer antipattern [46] or simply a newly introduced limitation due to new options, such as introducing new services to a given architecture that require additional overhead to use. Subtle underperformances introduced by a given change can introduce bottlenecks. However, this often goes undetected during the development and deployment processes, which could compromise the overall performance in the long run as the technical debt accumulates [36]. Finding the tipping point (or points) that resulted in underperformance becomes more tedious in later stages and relatively harder to mitigate [35]. Working around these introduced bottlenecks can introduce complexity in the architecture.

1.1. Motivation

With the increased usage of serverless applications spanning a wide variety of industries ranging from chatbots and IoT to machine learning [20], it became more important to study and understand the different performance trends in different ways. This helps us understand how the different FaaS platforms behave and work. Learning how FaaS

works can help us find new ways to utilize this new paradigm and potentially make it more efficient, secure, or reliable.

The main premise of FaaS and serverless architectures is that they are code snippets that request computing resources on-demand. It is interesting to observe how FaaS based applications perform under different loads and mature over time. This is particularly interesting given that FaaS is still in its infancy, which raises concerns about how the aforementioned FaaS based applications will perform in the long run.

1.2. Problem Statement

This thesis addresses the question of how different changes in a serverless architecture impact its performance during the development and deployment history. To the best of our knowledge, few tools address this question for FaaS and serverless architectures, such as the pipeline proposed by Vitalii et al. in their work [29]. Furthermore, there does not seem to be enough research on the topic despite the increased interest in FaaS and serverless architectures. Building a framework that captures the change in performance with every change made to a target serverless application not only helps in capturing the different performance trends to further study how a serverless application matures, but it also addresses the following:

- **Bottlenecks**: The framework, by design, would capture bottlenecks and help to pinpoint the exact change that caused it once it is introduces, which should save significant time that would normally be spent tracing the root cause of the performance dip.
- **Capturing Errors**: While having extensive testing for an application can prevent most changes that break the application from making it to the live system, this is not always the case. Reasons such as being unable to afford the live system replication or a slight difference in the configuration in the different stage environments can result in broken code being pushed to the live system. The framework proposed in this thesis would add an additional layer to capture such errors.
- **Vendor Lock-in**: Since the framework is built on top the monitoring framework from FDN (§2.2), which was built specifically to enable heterogeneous development in different ways, it supports most FaaS platforms that currently exist, thus not forcing the users into a vendor lock-in.

1.3. Contributions

This thesis aims to develop a framework that continuously monitors the performance change during a serverless application's development and deployment process. The framework also aims to compare how the different changes, such as introducing a new feature or modifying or deleting an existing one, impact performance over time. Our key contributions are:

- We develop and present a novel serverless regression framework called **RegX**, which aims to capture the change in performance after every commit made to a FaaS based serverless application. We develop **RegX** using Infrastructure As Code (IAC) to allow ease of deployment and distribution. To the best of our knowledge, this is the first work that addresses the question of how different changes in a serverless application impact its performance during the development and deployment history.
- As part of **RegX**, we extend the FDN-Monitor to work with AWS Lambda to collect various metrics of the individual functions (§4).
- We integrate **RegX** with a load testing framework based on K6 (for load testing the deployed application) and Grafana for providing the visual representation of the application's performance throughout the development cycle.
- Although our approach is generic and **RegX** can be easily extended to support other commercial and open-source serverless platforms, we demonstrate the functionality of **RegX** only with AWS Lambda (§2.1.3) on multiple serverless applications. We further present the performance results of the applications at various commit points (§5).

1.4. Outline

We introduce and discuss the different background topics needed to develop, deploy and run the proposed test framework in Chapter 2. We then discuss the different related works and how they inspired this work in Chapter 3. In Chapter 4 we discuss the implementation and deployment details of **RegX**: the serverless regression framework proposed in this thesis. Chapter 5 proceeds to break down and define the set of target test applications, metrics, and steps used to evaluate the set of target applications using **RegX**. The results of the evaluation from Chapter 5 are portrayed in Chapter 6 along

with the relevant graphically plotted results. We then discuss the results from Chapter 6 in Chapter 7 and the conclusions drawn from the work done in this thesis in Chapter 8.

2. Background

2.1. Cloud Computing

Before virtualization, developers had to spend significant time acquiring, maintaining, and creating the underlying physical infrastructure relative to the time spent on actual software development [27]. This process changed with the introduction of virtualization and containerization, allowing cloud computing to emerge.

Cloud computing is a model where cloud providers offer on-demand remote access to different resources with the ability to configure and modify them to fit their use. Examples of resources range from networks and servers to storage and fully managed services [18]. The cloud computing model removes a significant amount of overhead that would have been required to acquire, set up, and maintain these resources.

2.1.1. Cloud Providers

Cloud providers are generally companies that offer cloud computing services. Amazon's Amazon Web Services (AWS) and Microsoft's Azure Cloud are examples of companies providing cloud services. These providers usually provide a range of services from basic low-level Virtual Machines (VM) to Platform-as-a-Service (PaaS) and Function-as-a-service (FaaS) [37]. The services differ between providers. Despite the increasing number of cloud providers, we are only interested in Amazon Web Services (AWS) in this thesis as it is the most popular cloud provider [47].

2.1.2. Function As A Service (FaaS)

FaaS further extends the abstraction offered by serverless computing by introducing a newer layer of abstraction. This new layer allows the user to write event-triggered functions without maintaining the underlying environment setup, function, or scala-

bility. The cloud service providers handle running and scaling the triggered function based on minimal configuration input from the user, such as choosing the runtime environment from a list of predefined environments [47].

These functions have allowed developers to shift their focus to writing code snippets to implement the business logic without having to worry about managing the underlying infrastructure. As a result, a new paradigm for structuring applications emerged. The developers break the application into functions while primarily relying on FaaS and other serverless services, such as serverless databases. This is further discussed in the section §2.1.4.

2.1.3. Amazon Web Services (AWS)

Amazon's Amazon Web Services (AWS) is one of the most popular cloud platforms, with around 200 services provided [12]. Throughout this thesis, we are more focused on *Lambda*, *Simple Storage Service (S3)*, *Simple Notification service (SNS)* and *Dynamo DB*.

Lambda

AWS Lambda is a service that falls under the compute category of AWS services, which allows developers to write and execute functions without having to perform any server setup or management. Code snippets published on Lambda run on highly available infrastructure, fully maintained and managed by AWS. Lambda also scales according to usage, but can be configured to throttle or increase concurrency if needed. Lambda offers support for most of the common programming languages [14].

Simple Notification Service (SNS)

Amazon's Simple Notification service (SNS) implements the publish/subscribe design pattern, which is managed by AWS. The service allows different entities to publish different events for different topics to SNS. The subscribers are all notified in different ways, such as email and text messages. Simple Notification service (SNS) also accepts other AWS services as subscribers. SNS can be used to trigger different functions, such as AWS Lambda (§2.1.3) [15].

Simple Storage Service (S3)

Amazon's Simple Storage Service (S3) is a highly reliable object storage service offering secure and scalable data storage [11]. It is also possible to use S3 to trigger different AWS services, such as AWS Lambda, when a file is uploaded, modified, or deleted [7].

DynamoDB

Amazon Dynamo DB is a highly available and durable serverless NoSQL database that offers fast access speed by utilizing different access patterns [48, 10] and other features, such as using SSD for storage [49].

APIGateway

API Gateway is an abstraction analogous to a reverse proxy, which lies between a client and different back-end applications to direct requests accordingly and retrieve the response [44, 51].

AWS offers its own implementation of an API gateway that allows for building REST and web socket APIs while providing an added layer of security, monitoring, and scalability [8].

Boto3

Boto3 is a python implementation of the AWS SDK [2], which can be used to build applications utilizing the different AWS resources, such as storage services like S3 and compute services like lambda. Using Boto3, one can collect CloudWatch metrics and log insights for the various log namespaces.

Identity and Access Management (IAM)

Identity and Access Management (IAM) is a service that manages access between different entities, such as AWS console users and other AWS services. Identity and Access Management (IAM) achieves this by using user groups, roles, and resource access policies [13]. AWS offers predefined IAM configurations but allows users to

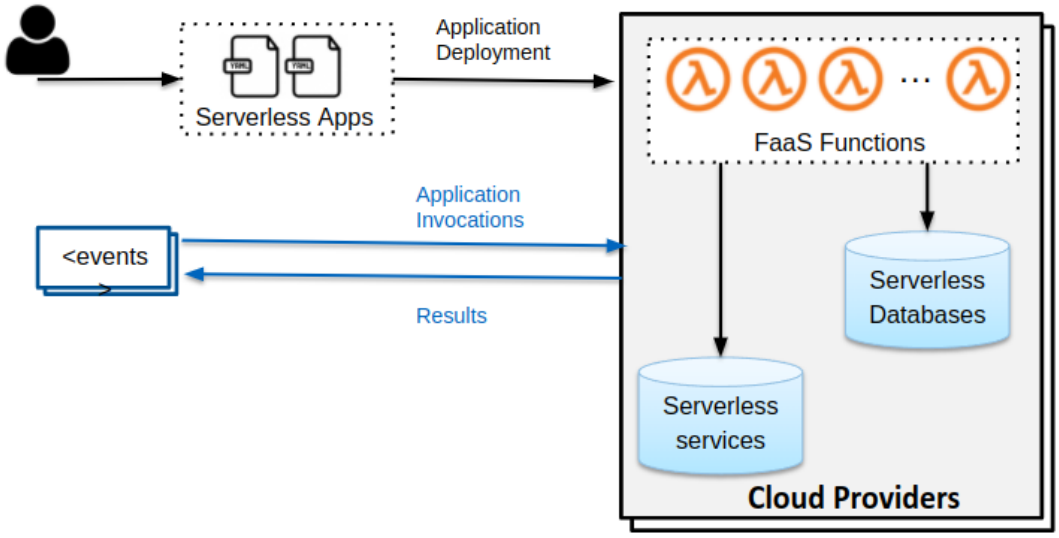


Figure 2.1.: A basic overview of a basic serverless application

create their custom configurations and policies. IAM everywhere in AWS, such as allowing lambda functions to access S3 buckets (§2.1.3).

2.1.4. Serverless Computing and Applications

Serverless computing is a model which takes advantage of the FaaS paradigm. The application logic in this model is broken down into functions, which respond to different events. In case an application uses a database, a serverless database such as Amazon's DynamoDB (§2.1.3). It is also quite common to use an API gateway (§2.1.3) for functions that should respond to HTTP requests (e.g., REST APIs). Figure 2.1 shows an example of what a serverless application could look like.

2.2. FDN

In their work, Jindal et al. [32] propose a solution that extends serverless computing to heterogeneous clusters. It achieves this by introducing different tools to allow deployment and monitoring, among other features, on different FaaS platforms based on a set of user-provided configurations. The different cross-platform clusters are

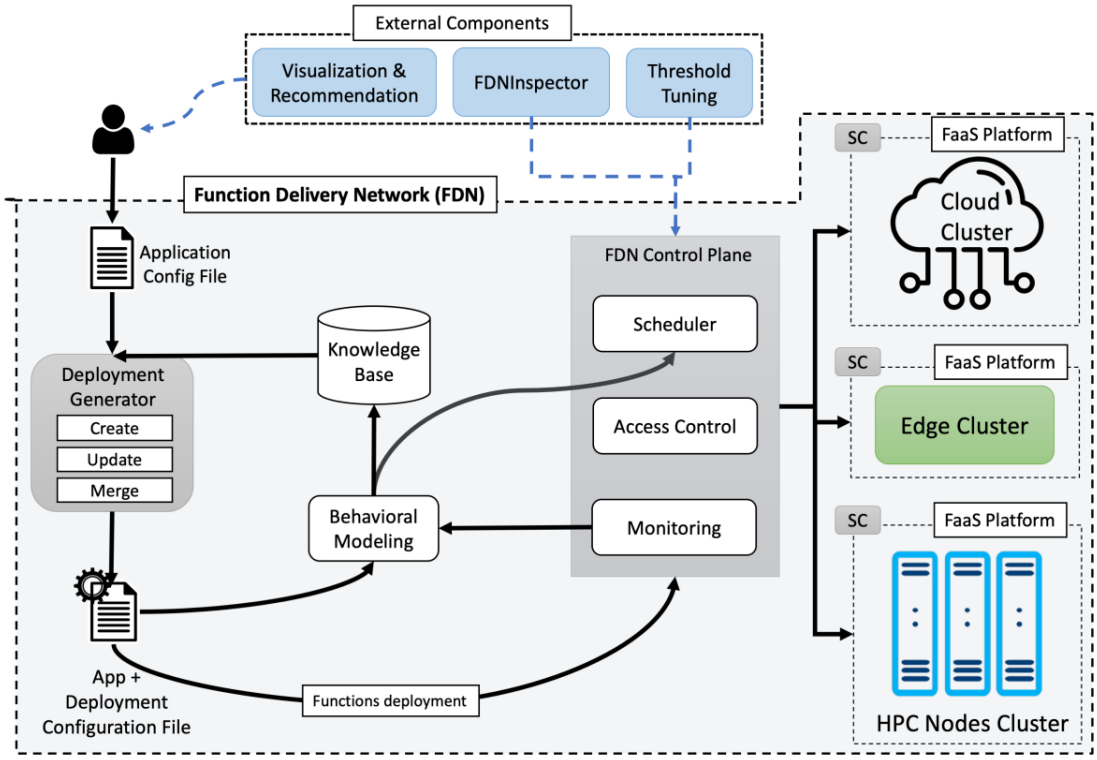


Figure 2.2.: An overview of a Function Delivery Network (FDN) architecture presented by Jindal et al. [32]

coupled through the control plane component in Figure 2.2, allowing the developers to quickly deploy serverless functions across other platforms with minimal platform-specific configuration.

Within the previously mentioned control plane module lies a monitoring sub-module, which collects different metrics for heterogeneous serverless applications. For example, the monitoring module can be used to fetch the average execution time for a lambda function hosted on AWS through the collected CloudWatch logs and log insights. We extract this module and utilize its capabilities throughout this thesis.

2.3. Load Testing

One way to obtain insights into an application is to see how it performs under different loads. This ideally simulates real-life user interactions with similar services that already exist. The application is closely monitored during this process to collect various metrics about an application.

2.3.1. K6

Unlike most traditional load testing tools, which have a graphical user interface (e.g., Jmeter), K6 allows users to write test scripts in JavaScript files [40]. This makes it a more powerful tool for power users, as it can generate dynamic load testing logic using JavaScript capabilities. K6 also uses less memory, enabling deploying it in most virtual machines [40]. Once K6 completes running a script, it outputs the different metrics it collected to the terminal. K6 also supports exporting the results to various destinations, such as Influx DB.

Virtual Users (VUs)

Virtual Users simulate how real users would interact with a website, defined as a concurrent connection to a service, similar to how a user would interact through a browser [33]. The number of requests made by virtual users can be tuned to allow for more flexible testing for APIs.

2.4. DevOps

DevOps is a model aiming to bridge two previously different teams within an organization. The teams are development and operations, derived from the word DevOps [4].

DevOps aims to achieve the goal of integrating these two teams by automating the different processes within development, deployment, and automated monitoring for applications and infrastructure. As a result, these different tasks are no longer the responsibility of any isolated team within an organization on its own, but rather a collaborative effort comprised of cross-functional teams [19]. The paradigm introduced by DevOps increased the velocity of delivering value continuously.

Throughout this thesis, we use different DevOps tools, such as "Continuous Integration and Development" §2.4.2 and *Infrastructure As Code (IAC)* (§2.4.3), to help with the development and ease the process for any upcoming work to work created in this thesis.

2.4.1. Gitlab

GitLab is a git repository manager with a web interface and a wide range of tools. These tools help developers develop and ship their applications to different hosting platforms [25]. A subset of these tools is the DevOps tools and services offered by GitLab, which are discussed in more detail in the following sections §2.4 and §2.4.2.

2.4.2. Continuous Integration and Development (CI/CD)

As previously mentioned, automation and continuously delivering changes to a code base are some tasks required by DevOps to increase development velocity (§2.4). CI/CD is a way to help achieve these goals, which introduces the ability to automate the workflow necessary at different stages [45].

GitLab offers its implementation of CI/CD, where the configuration and instructions required are provided by the user in a file named `.gitlab-ci.yaml` [26]. The files describe the stages that make up a pipeline, the different jobs within a stage, and the relation between stages if one exists (e.g., the test stage relies on the build stage).

2.4.3. Infrastructure As Code (IAC)

Infrastructure As Code (IAC) is a way to automate infrastructure, allowing users to write the components that make up the underlying infrastructure as code, the same way a developer writes code. This is one of the basic principles of the DevOps model [5]. Infrastructure As Code (IAC) allows for quick infrastructure deployment and redeployment across different vendors or regions if needed. It also makes the experience more homogeneous than alternative ways of achieving it through scripts [5, 39].

Cloud Development Kit (CDK)

Cloud Development Kit (CDK) is an open-source framework that allows users to write infrastructure as code using popular languages, such as TypeScript, Python, Java, and .NET. This allows Cloud Development Kit (CDK) users to take advantage of the existing support for these languages when defining their infrastructure [1].

2.5. Time Series Database

To obtain meaningful insight into the performance of a given system, we usually monitor its performance consistently throughout a given period. To get the most insight into a data set, we need to define the metrics of interest and the rate at which the data is collected (e.g., every 10 seconds). This type of data is often referred to as time-series data [42].

While time-series data can theoretically be stored using most databases, it is usually stored using time-series databases. This is because time series databases are specifically designed for this type of data. Time series databases take advantage of the characteristics of this data, thus having more optimizations in mind compared to other kinds of databases. For example, time-series databases optimize storing data points by taking advantage of the fact that prefixes of the Unix timestamp are usually repeated for consecutive data points and only storing the common prefixes for such points. Other databases, such as relational databases, do not take advantage of the different features present in time-series data. This lack of time-series-specific optimization results in sub-optimal data storage and retrieval.

Time-series databases are collections of values, which are periodically calculated or retrieved [23]. The data stored in time series databases are typically extensive. Throughout this thesis, we will use Influx DB (§2.5.1) to store time-series events and metrics for the different serverless functions [41].

2.5.1. InfluxDB

InfluxDB is an open-source implementation of time series databases. It was built on top of an open-source core that was implemented explicitly to create a time series database rather than modify an existing database to accommodate time series data, which made different optimizations possible [28].

InfluxDB integrates with different tools that allow data collection, monitoring, and different optimizations. However, we are more concerned with the data collection and storage aspects of influx DB in this thesis.

2.6. Monitoring

In this section, we discuss the different monitoring tools that will be used throughout this thesis.

2.6.1. CloudWatch

CloudWatch is an AWS monitoring tool that integrates seamlessly with the different AWS services, like previously mentioned, which collects different metrics for each service and resource [9]. CloudWatch also allows for building and creating different types of dashboards and setting up alarms, among other features.

CloudWatch, like many AWS services, can be reached via an API[1] which makes it possible to collect data from an external source for further processing or visualization if needed. Throughout this thesis, we collect metrics captured by CloudWatch using the API provided by AWS.

2.6.2. Grafana

Grafana is open-source software that offers multiple visualization tools for time series databases, which can be used to create dashboards from various data sources. It also allows users to execute their queries, set alarms, and create annotations [41]. Figure 2.3 shows what a dashboard in Grafana would look like.

[1] https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/Welcome.html

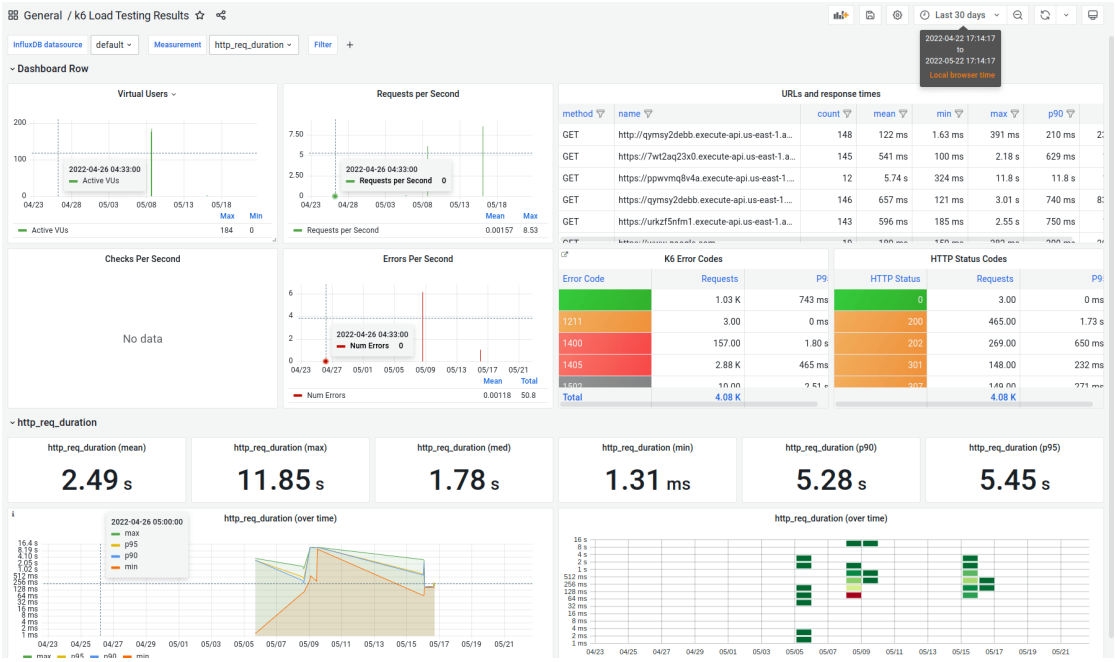


Figure 2.3.: An example of a Grafana panel with different dashboards showing different metrics collected by K6 §2.3.1.

3. Related Work

3.1. Automatic performance monitoring and regression testing during the transition from monolith to microservices

In their work, Andrea Janes et al. introduce an approach to monitor the negative impact on performance over time throughout the process of breaking down a monolith application into an equivalent microservice application [30] . Their goal was to attempt and avoid the adverse effects on performance during the transition from a monolith architecture to a microservice architecture. For example, suppose the new microservice architecture underperforms by a certain threshold. In that case, the developers can opt to shift their focus on the degradation in performance to avoid further impact. To achieve this, Production and performance application monitor (PPTAM), a testing-based application monitoring tool, was modified to monitor all user interactions. PPTAM was also used to store the response time metric for the existing monolith architecture and the new monolith system. The tool then plots both performances and alerts the developers if the performance of the new microservice architecture drops past the threshold set up by the developer. An alert is raised for the developers. The tool also offers a user interface for interested stakeholders to view the different graphs mentioned above. Figure 3.1 shows an example of the tool setup.

Throughout this thesis, we seek to create a performance monitoring framework with the aim of tracking performance degradation in serverless and FaaS-based applications, similar to the tool discussed here. We benefit from the general idea and the approach introduced by this work.

3.2. The Serverless Application Analytics Framework

In their work Robert et al. discuss the impacts of different languages on the performance of FaaS, specifically AWS Lambda [17]. To achieve this, they introduce Serverless

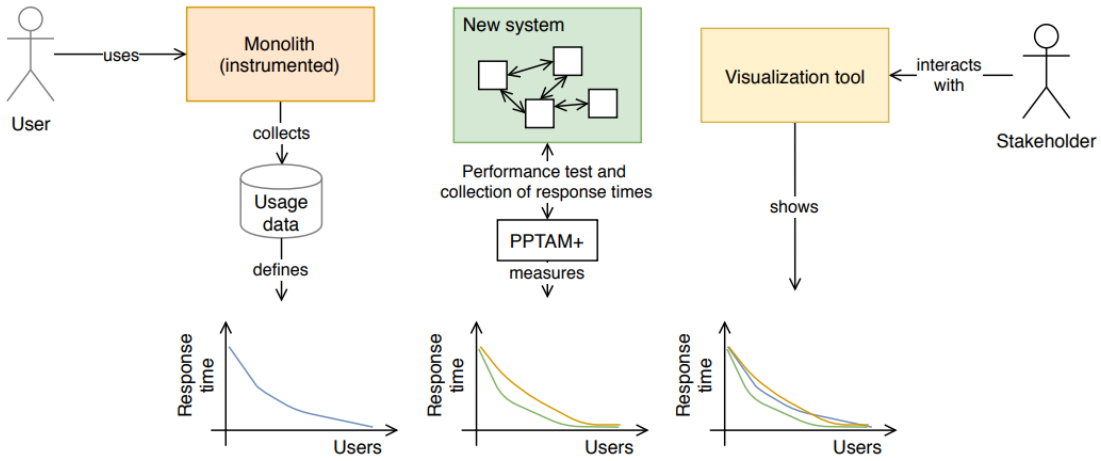


Figure 3.1.: Performance monitoring design presented in [30])

Application Analytics Framework (SAAF). SAAF collects forty-eight different metrics, including, but not limited to, memory, total execution time, and latency, which are compared across other FaaS platforms.

In this thesis, we build a similar tool that collects a subset of the metrics collected in work proposed by Robert et al. in [17]. However, our serverless regression framework performs that in an automated manner. It aims to compare the change in performance throughout the different iterations of the application, regardless of the programming language or the FaaS platform.

3.3. Implementation of a DevOps Pipeline for Serverless Applications

Vitalii et al. discuss the impact of serverless applications on the DevOps pipeline and propose a CI/CD and monitoring pipeline [29]. In their work, they research how the serverless model impacts the DevOps role and processes. While they suggest that the serverless application model introduces useful DevOps tools, it also presents some monitoring challenges. These challenges are due to generally having tight coupling with other cloud services and no access to the underlying infrastructure for debugging. Their research findings introduce a GitLab pipeline, which is automatically triggered when a change is pushed. A runner then picks up the job and runs the script for the

given stage. If the right stage is triggered, end-to-end testing is triggered automatically.

In this thesis, we build a CI/CD pipeline similar to the one proposed in this work to automate the deployment and load testing processes. However, we modify it to fit our framework. We use the GitLab CI/CD pipelines to deploy our developed framework. We also use GitLab's CI/CD pipelines to upload test scripts on the target application and to trigger load testing and cloudwatch metric collection.

3.4. FaaS optimization

Most previous work [21, 38] focus on working around the cold start problem for serverless applications. This might be useful to overcome such a well-known problem that stems from the current design of the FaaS model offered by most providers. However, it does not consider any performance issues resulting from the change in the complexity of a FaaS-based serverless application as it matures and how the increased coupling with other serverless services impacts performance.

In this thesis, we aim to capture any potential sub-optimal performance resulting from design changes to a FaaS based serverless application that results from the serverless application maturing over its lifetime. This tackles the performance issues for lambda from a different perspective by capturing performance issues as they occur.

4. RegX: Serverless Regression Framework

This chapter covers the implementation details of the Serverless Regression Framework called **RegX**[1], which is an open-source automatic test framework that automatically performs regression testing, collects metrics, and publishes the results to grafana dashboards upon every change introduced to a code repository. This framework represents one of the main contribution for this thesis.

RegX is created as a serverless application using Amazon's CDK (§2.4.3) with Typescript[2], making it easy to deploy. To further ease deployment, we configured the framework to use different DevOps tools, such as GitLab's CI (§2.4.2), to automatically deploy any changes made to the framework. Throughout this chapter, we will discuss the system architecture, different technical design choices, and implementation details of every component within **RegX**.

4.1. System Architecture

The architecture of **RegX** is shown in figure 4.1 and consists of three main parts: 1) *A GitLab* repository, which holds the code base and acts as a CI/CD tool to automate the deployment process and upload the test script, 2) the core *test runner data collector component*, which runs uploaded test scripts and collects CloudWatch (§2.6.1) metrics, and 3) a *visualization and storage component*, which stores the collected time-series (§2.5.1) metric data and offers different dashboards to view this data. The Figure 4.1 also includes a sample serverless application, which acts as a test subject but is not a part of the framework.

The core components in this framework are robust enough to work with other elements that offer similar interfaces. For example, GitLab can be replaced with any repository management tool and a service that provides CI pipeline tools, such as Travis CI or

[1] https://gitlab.com/hady1100/Framework-for-Performance-Regression-Evaluation-of-Serverless-Applications

[2] https://www.typescriptlang.org/

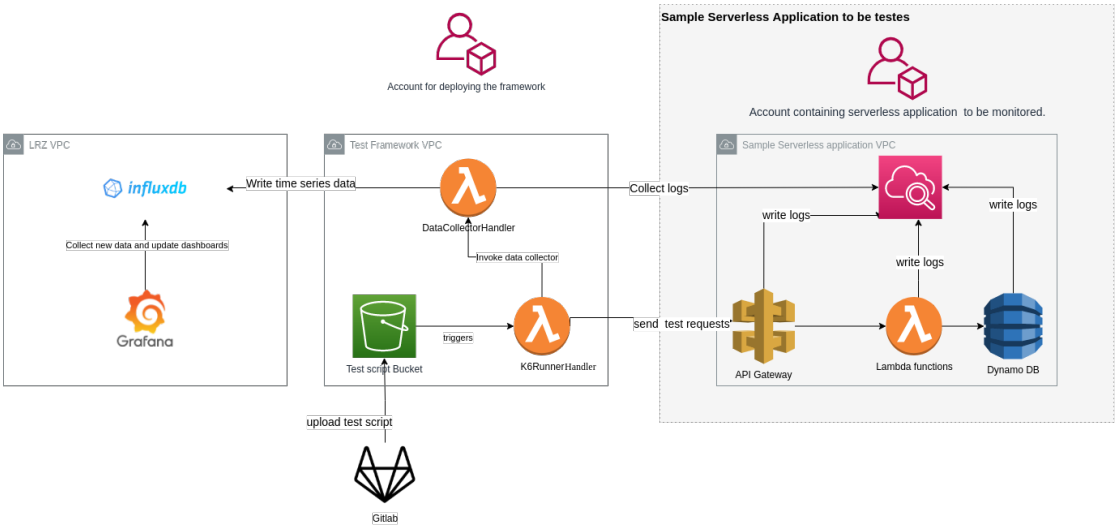


Figure 4.1.: A high-level overview of the different components that make up the infrastructure for the test framework

GitHub and GitHub Actions[3]. The *test runner and data collector component* only expects a K6 test script to be uploaded to the correct bucket and can export the results in different formats. The visualization component is the only component that could require some parameter tuning to display the correct data on the dashboards.

The general flow starts when a developer pushes changes to a repository of a serverless application that has a K6 test script and CI/CD configured. This action triggers the CI pipeline, which deploys the new changes made to the application on the relevant cloud platform and proceeds to upload the K6 test script to the *testscriptbucket*. The Simple Storage Service (S3) bucket then triggers the *K6RunnerHandler*, which runs regression tests using the previously mentioned K6 test script against the target serverless application. Once the regression testing is complete, the *K6RunnerHandler* pushes the collected user-centric metrics (§5.2) to *influx DB* and invokes the *DataCollectorHandler*, which collects the different *CloudWatch* (§2.6.1) metrics produced as a result of running the K6 script and pushes it to *influx DB* as well. Further details regarding this component can be found in (§4.2.2).

[3]https://github.com/features/actions

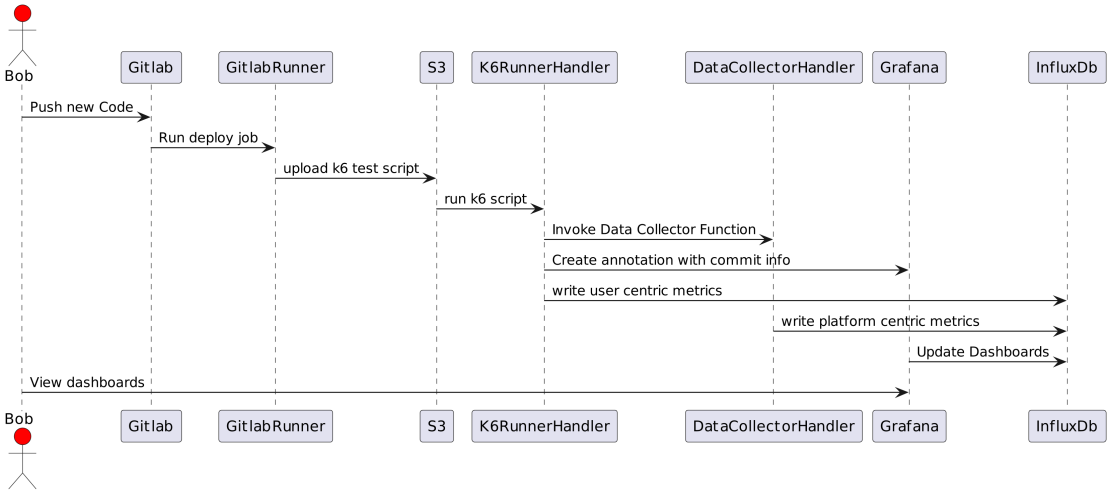


Figure 4.2.: A sequence diagram of **RegX** showcasing the entire flow of the framework.

4.2. RegX Components

This section breaks down the different components that make up **RegX**. We start by explaining the role of *GitLab* as a repository management tool and a CI/CD tool. We then proceed by discussing the *Testing and Data Collector*, which is responsible for running tests and data collection, followed by the *Data Vsizualization and Storage* Unit, which presents the collected data graphically.

4.2.1. GitLab

This section explains how the different features and tools offered by GitLab are used in the implementation of **RegX**. We consider GitLab to be its own component, which is used to automate the deployment of **RegX** and trigger the *Testing and Data Collection Component*.

GitLab, in this architecture, serves as a repository manager and a CI pipeline for the serverless regression framework's source code and any potential target applications we expect to be tested. It is important to note that any other repository manager and CI tool can replace GitLab in this architecture.

GitLab CI pipeline is made up of logical steps called *stages*. Each stage can be broken down into multiple jobs, where a job executes a given set of commands. Each job

can independently use any container image as a base for executing all following commands. This allows the commands to run in any containerized environment, such as an environment with NodeJS or python configured. The GitLab CI pipeline is configured through a YAML file named `.gitlab-ci.yml` and has to exist at the repository's root [24]. The CI pipeline used for the framework is configured to be comprised of 3 stages: 1) deploy any changes made to the infrastructure, 2) run a sample test to ensure the framework is working, 3) manually trigger the deletion of the deployment stack for the entire test framework infrastructure when the prototyping is over.

As previously mentioned in this chapter, the serverless regression framework expects the user to have a K6 test script in the root of their repository that is uploaded upon every deployment-ready change. The CI pipeline is configured to achieve automatic uploads for the K6 test script, using the sample code in Code listing 1.

```
1         uploadTests:
2           stage: test
3           image:
    ↪  registry.gitlab.com/gitlab-org/cloud-deploy/aws-base:latest
4           script:
5             - aws -version
6             - ls
7             - aws s3 cp ./test.js s3://serverlessscriptbucket/
```

Listing 1: An example .gitlab-ci.YAML configures a job within a test stage in the GitLab Ci pipeline.

The code snippet includes a job with the name *uploadTests*, which belongs to a stage called *test*. The base image for the job is the *aws-base*, provided by GitLab itself, among many other images officially hosted by GitLab on their container registry [4]. The script includes a list of scripting commands to run in a given order. In this case, it includes the commands required to upload the test script to the *testscriptbucket*. For this to work properly, the user needs to configure GitLab CI by providing the correct IAM (§2.1.3) role or credentials to locate and upload files to the S3 (§2.1.3) bucket[5].

[4]https://docs.gitlab.com/ee/user/packages/container_registry/
[5]https://aws.amazon.com/blogs/apn/using-gitlab-ci-cd-pipeline-to-deploy-aws-sam-applications/

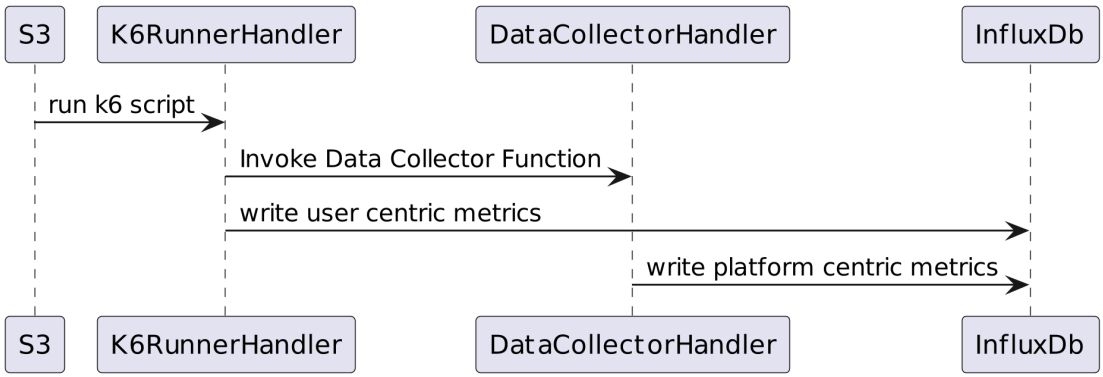


Figure 4.3.: A sequence diagram describing the flow of the *Testing and Data collection Unit*.

4.2.2. Testing and Data Collector Unit

This component handles running regression tests against different serverless applications intended to be tested. It then collects the different types of metrics produced from testing the serverless target application. Figure 4.3 shows a close up of this component.

The component can be conceptualized as the back-end of the test framework. It can be further broken into the following components: 1) an S3 bucket to hold the test script that will be run against the target application, 2) a lambda function that runs the test script and collects user-centric metrics, and 3) a data collector lambda function which collects platform-centric metrics. The following subsections discuss the different components that make up this unit in greater detail.

K6 Runner

The K6 runner is represented by the **K6RunnerHandler** in the Figure 4.1 and contains a simple wrapper, which is written in Javascript, over the K6 CLI[6] tool. The k6 wrapper is a modified version of the code discussed in [50]. Upon file upload to the **testscriptbucket** S3 bucket, the **K6RunnerHandler** is triggered. The lambda function then fetches the test script file from the S3 bucket and saves it locally. The appropriate IAM roles and permissions (§2.1.3) are assigned to the S3 bucket and are defined using CDK (§2.4.3). The IAM roles allow S3 to trigger lambda functions and allow lambda

[6]https://k6.io/

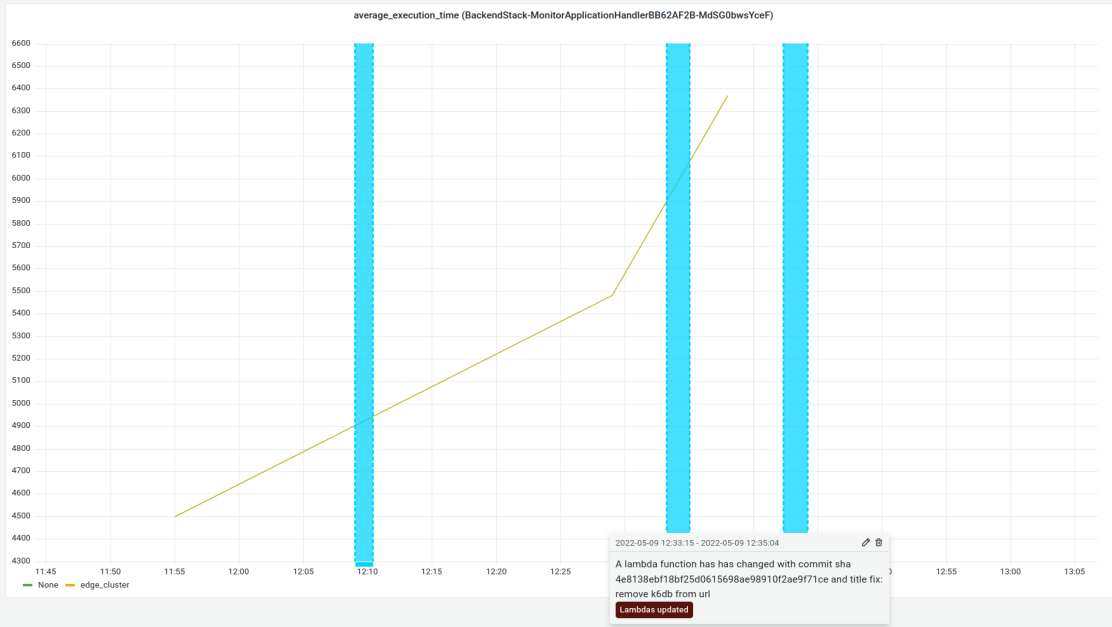


Figure 4.4.: An example of an automatically generated annotation for running different traces. The metric shown in this example is for the average execution time.

functions to access S3.

With the test file saved locally, the function creates a sub-process that runs the K6 CLI tool to run the test script, saving the output in Influx DB. Once the sub-process completes execution, the function creates an annotation with the commit number using Grafana's API. The *K6RunnerHandler* function then invokes the *DataCollectorHandler* function and responds with a success message (or a failure message if something goes wrong). The annotation created spans the entire execution time of the K6 test script, making it easier to identify the impact of every commit on the different performance trends. Figure 4.4 shows an example of this annotation.

Data Collector

The data collector component is responsible for collecting the different platform-centric metrics produced from executing different FaaS functions within a serverless application. This section will discuss what the component does and its implementation details.

When a serverless function with proper monitoring setup[7] is triggered, the FaaS platform hosting this function collects different platform-centric metrics. For example, when an AWS Lambda function is triggered, different metrics, such as execution time and memory usage, are collected by Amazon's CloudWatch (§2.6.1). The component discussed in this section can then be used to retrieve these metrics from CloudWatch (§2.6.1), or other similar services for different FaaS platforms, to aggregate all of this data in a central place. To store all the collected metrics, we use InfluxDB[8], a dedicated time series database that is well maintained and highly scalable.

The component discussed in this section is a modified version of the work proposed by Jindal et al. in [31], which can be found in the FDN-Monitor repository [9]. While working on this component, various contributions were made to the FDN-Monitor repository. The component discussed in this section is written in python and supports the following FaaS platforms: AWS, Google, Open FaaS, and Open whisk. Throughout this thesis, we focus more on AWS. However, most concepts and functions that the AWS data collector is based on are similar across the different data collectors. Figure 4.5 shows the UML diagram describing the structure of the other data collectors with AWS and Google data collectors filled with showcasing the similarity above.

The module takes advantage of different python libraries and SDKs, such as boto3 (§2.1.3) for AWS. These libraries and SDKs take advantage of the APIs provided by different FaaS platforms and cloud providers to collect different types of metrics. The general flow of the data collector module 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 influx DB, the credentials required to connect to the FaaS platform, and the cluster type (e.g., AWS). Upon reading the environment variables, the main function instantiates the correct type of data collector. For example, if the cluster type is set to AWS, an instance of *AWSDataCollector* is instantiated with the correct AWS account credentials. In the case of AWS, the different *collect_<metric_name>()*, from Figure 4.6, functions are then called asynchronously. The primary function responsible for collecting the platform-centric metrics from AWS can be seen in the code listing 2. Once the metrics are collected, they are combined into one data frame. The collected metrics are filtered during the combination process to remove any empty results. The *collect_data_from_logs()* function is then called, which collects additional metrics provided by log insights - A feature for parsing logs provided by AWS. This allows for further platform-centric metric

[7] Some platforms automatically setup and configure monitoring, such as AWS, while others require that the user configure monitoring.

[8] https://www.influxdata.com/

[9] https://github.com/Function-Delivery-Network/FDN-Monitor

extraction, such as *billed duration* and *maximum Memory Used*. Responses from log insight queries are then processed and combined once again with the data frame from the previous steps, based on the timestamp. Further post-processing, such as converting time columns to seconds, calculating mean values, and column renaming, take place. This is done to prepare the data for influx DB and to match the names of the different Grafana dashboard variables to maintain cross-platform compatibility.

```
async def get_and_convert_data_frame(self, start: int, end: int, stat_type: str,
    feature_col_name: str, save_feature_col_name: str) -> DataFrame:
    """ Get and Convert the timeseries data values to a dataframe .
        Args:
            start: Integer - A timestamp, where the query range should start
            end: Integer - A timestamp, where the query range should end
           feature_col_name: String - the name of the feature column name
        Returns: DataFrame - Query result as DataFrame - with columns: 'timestamp', 'action', 'region', 'memory',
            'feature_col_name'
        """
        stats = self.cloudwatch_client.get_metric_data(
            MetricDataQueries=[
                {
                    'Id': 'metric_data',
                    'Expression': "SEARCH('{" +
                                  self.metric_namespace+",FunctionName} "
                                                        "MetricName=\""
                                                        + feature_col_name+"\"', '"+stat_type+"', "+str(self.period)+")",
                    'ReturnData': True
                },
            ],
            StartTime=start,
            EndTime=end,
            ScanBy='TimestampDescending'
        )
        values = []
        for record in stats['MetricDataResults']:
            for idx, inner_record in enumerate(record['Timestamps']):
                values.append(
                    {"action": record['Label'], "timestamp": inner_record, save_feature_col_name: record['Values'][idx]})
        return DataFrame(values)
```


Listing 2: A code snippet from the *DataCollectorHandler* that handles the primary logic for retrieving CloudWatch (§2.6.1) metrics using boto3 (§2.1.3)

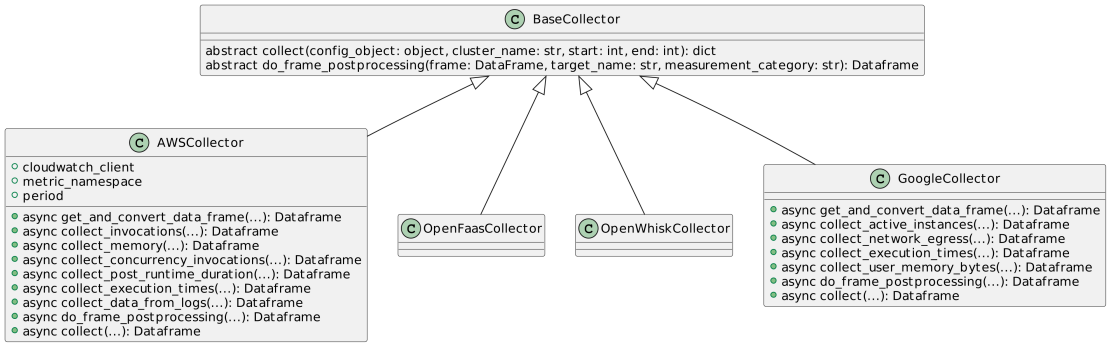


Figure 4.5.: A simplified UML showcases the different data collectors for the different platforms.

A wrapper had to be introduced to get the data collector module from FDN-Monitoring, working on lambda, before integrating it into **RegX**. This wrapper acts as an entry point to comply with lambda requirements and acts as a lambda handler. A few changes, such as updating queries and modifying retrieved fields, were made before finally containerizing the module using docker to include all environmental requirements and dependencies.

All function details were defined using CDK (§2.4.3), which is An open-source development framework that allows developers to define their cloud resources for AWS [1]. All necessary IAM roles and permission (§2.1.3) required to deploy and modify the function were configured and assigned to the CI pipelines, while the permissions to invoke the function were assigned to the **K6RunnerHandler**.

4.2.3. Data Visualization and Storage Unit

The unit discussed in this section is responsible for storing the collected metrics in a time-series database and providing the user with a visual representation of this data.

This component relies on two open-source projects: Grafana (§2.6.2), which is a data visualization tool, and Influx DB (§2.5.1), which is a time series database. Influx DB is responsible for storing the data collected by both the *K6RunnerHandler* and the *DataCollectorHandler* functions from the test runner and data collector component from §4.2.2, which can be seen in Figure 4.1. Grafana reads this data and updates two dashboards that were explicitly selected for the work conducted in this thesis.

Influx DB

Throughout this thesis, we use influx DB, a database built specifically to handle time-series data with high efficiency and scalability (§2.5.1). InfluxDB is responsible for persisting platform and user-centric metrics produced by the *Testing and Data Collection Unit* (§4.2.2). InfluxDB is also used by Grafana (§2.6.2) as a data source to plot and update the different data points.

Grafana

Grafana is an open-source data visualization tool [41], which can collect data from different sources to create different dashboards. Grafana supports various query languages to obtain the desired data points from the relevant data source. We used different dashboards to visualize the aforementioned metric data produced by the serverless regression framework. In this section, we will discuss the different dashboards we used.

Platform Specific Dashboard

This dashboard aims to represent the platform-specific metrics collected by the data collector unit §4.2.2. This includes the following metrics:

- **Invocation**: The number of times the function got invoked.
- **Average execution time**: The average time it takes to execute the code snippet contained within the lambda function.
- **Average memory used**: The average memory resources needed to execute the code snippet contained within the lambda function.
- **Concurrent Executions**: The sum of concurrent lambdas required to fulfill concurrent requests.

The dashboard template from the *FDN-Monitoring* repository [10] was used to display the platform-specific data [22]. Throughout this thesis, different contributions and upgrades have been made to the *FDN-monitoring* repository to make it more robust and to test some of its features more thoroughly. Since the dashboard builds on top

[10]https://github.com/Function-Delivery-Network/FDN-Monitor

of the *FDN-Monitor* repository, a part of Function Delivery Network (FDN) that was previously mentioned §2.2, it is robust enough to handle platform-specific metrics collected from different FaaS (§2.1.4) providers, not just AWS Lambda.

The dashboard takes advantage of the variable template feature offered by Grafana to allow it to display the same dashboard for each function within a serverless application. An overview of the dashboard discussed in this section can be seen in the Figure 4.7 and its JSON definition for Grafana [22].

User Specific Dashboard

This dashboard aims to represent the User-centric metrics collected from running the K6 test script by the **K6RunnerHandler** function. This dashboard includes the following metrics: Number Virtual Users (VUs), URL response times, errors rate HTTP duration with heat maps and summaries to show an appropriately detailed overview. This can be seen in Figure 2.3.

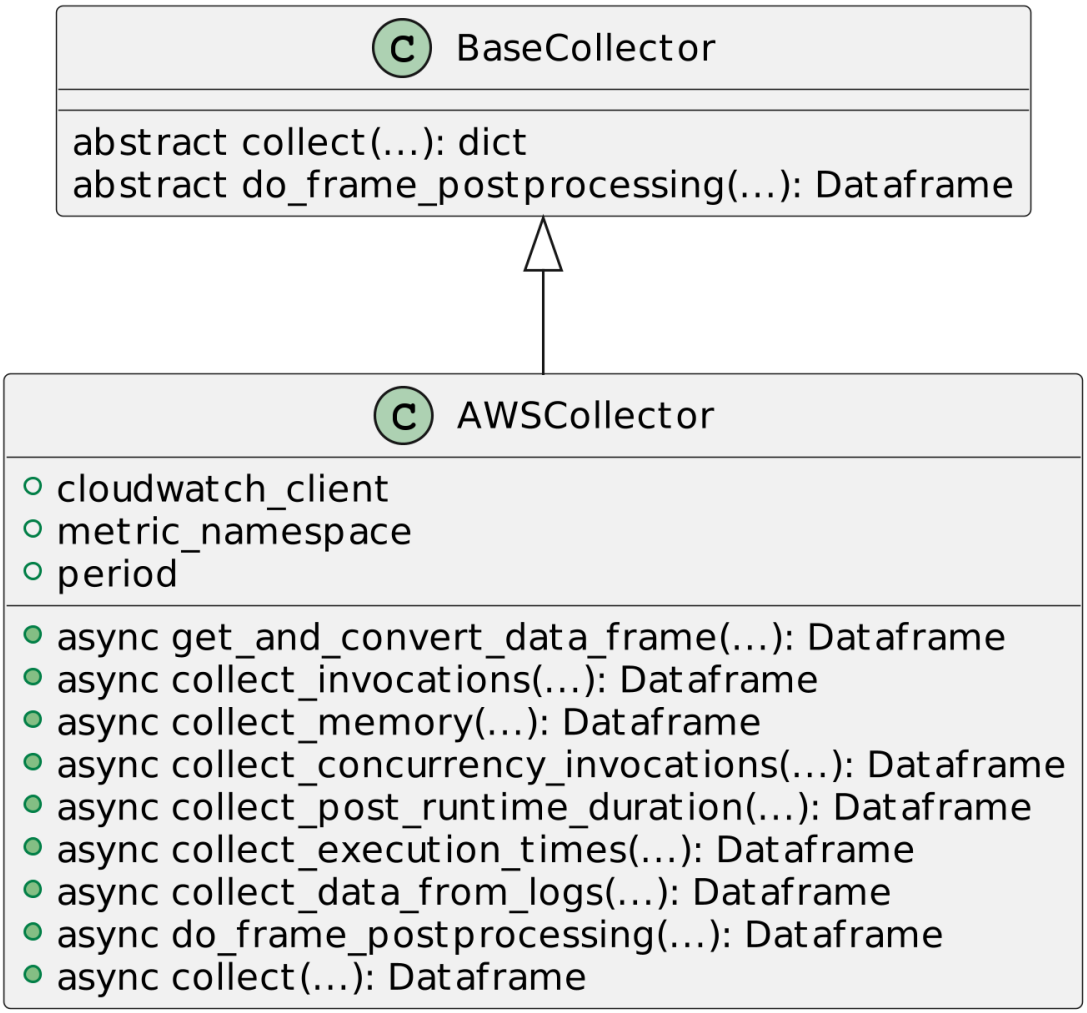


Figure 4.6.: A simplified UML showcasing the **AWSCollector** class details.

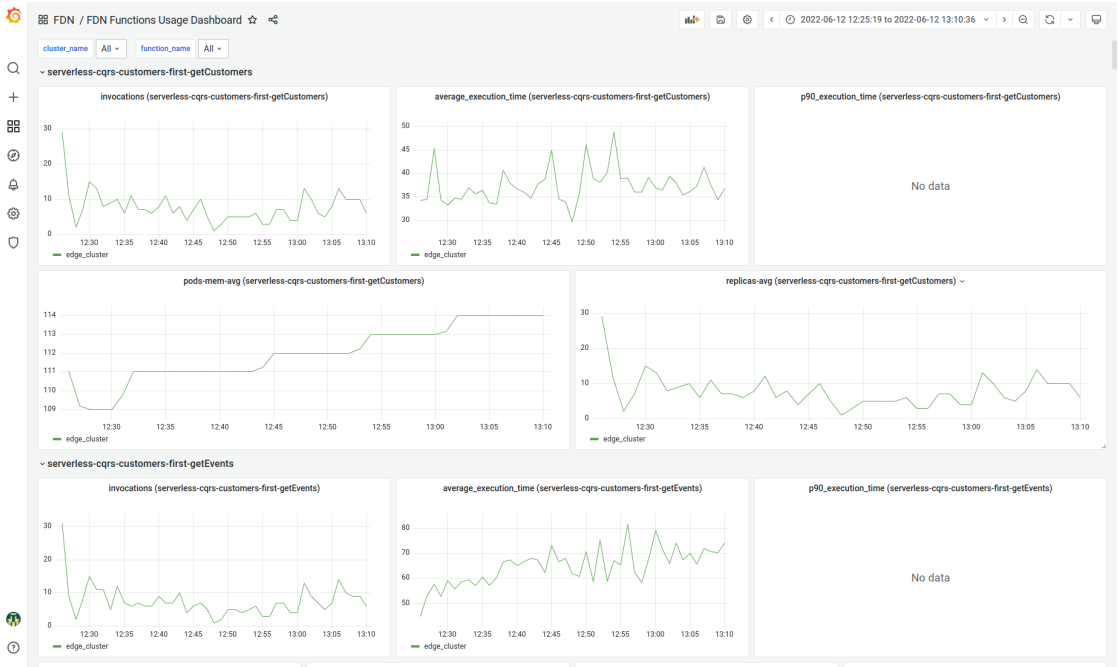


Figure 4.7.: An overview of the Grafana dashboard showing different platform-specific metrics.

5. Evaluation Settings

In this chapter, we discuss the methodology used to test a set of serverless applications using the *Serverless Regression Framework* (§4). To properly showcase the usecase of *Serverless Regression Framework*, we need different sample serverless applications with a realistic development history to be tested at different points in their development lifetime. Therefore, We first introduce the benchmarked applications that we use to test **RegX** in §5.1. Following this, we present the performance metrics collected from AWS Lambda in §5.2.

5.1. Benchmark Serverless Applications

We define eligible applications as the set of serverless applications with at least five or more commits with incremental feature development or bug fixes. Serverless application with too few commits, i.e. below 5 or with all the code pushed in one commit, are not eligible to be chosen for testing. After careful consideration, the following two applications are chosen and deemed eligible for testing.

5.1.1. Serverless Restaurant Application

The first application is a serverless-restaurant[1] application, and its architecture can be seen in Figure 5.1. It consists of 3 main similar components, which simulate a specific set of tasks relating to one of the following entities: Orders, Menus and Customer. For example, the component responsible for tasks related to *customers*, is responsible for registering a new customer.

Each of the components consists of an *API Gateway*, which handles incoming HTTP request. This allows triggering lambda using HTTP requests. The components also contains different *Simple Notification Service (SNS)* (§2.1.3), that notifies all subscribers

[1] https://github.com/vvgomes/serverless-restaurant

when a publisher publishes a message to these topics. *SNS* topics act as a trigger for the subset of the lambda functions that are not triggered by the API Gateway. The set of functions that use *SNS* topics as triggers are the ones responsible for inserting and updating the different *Dynamo DB* table. All the aforementioned resources produce different logs, which are pushed to *cloudwatch*. However, we are only interested in the logs produced by the lambda functions. The applications are deployed with the serverless framework, with each component deployed separately in its own configuration file. An example of a single component can be seen in Figure 5.1.

Upon inspecting the different commits for this application, we notice that the developers started with a simple architecture that did not include all the aforementioned resources in figure 5.1. The application started with a single working component for the *Customer* related requests, which included an API Gateway and a single lambda function. The developers then proceeded to introduce more lambda functions and SNS topics to trigger the new lambda functions, as well as Dynamo DB tables to persist the different data. This introduced more complexity and increased coupling with different serverless services. The final result is an implementation of the Command and Query Responsibility Segregation (CQRS) architectural pattern [52], which segregates the read and update operations for a given storage layer.

5.1.2. AWS Lambda Typescript

The second application[2] aims to provide a working example of how a serverless application would be implemented in typescript from the perspective of the developers who created it.

Similar to the previous serverless application (§5.1.1), this application is packaged and deployed using the serverless framework. However, it uses a simpler architecture, which can be seen in Figure 5.2. The application consists of a single API gateway and a lambda function. Over time, the developers introduce a few more lambda functions, the implementation details of which are not relevant to this section. Despite the huge number of commits and contributions to this application, its design remains relatively simple throughout its lifetime compared to the application discussed in the previous section.

[2]https://github.com/balassy/aws-lambda-typescript

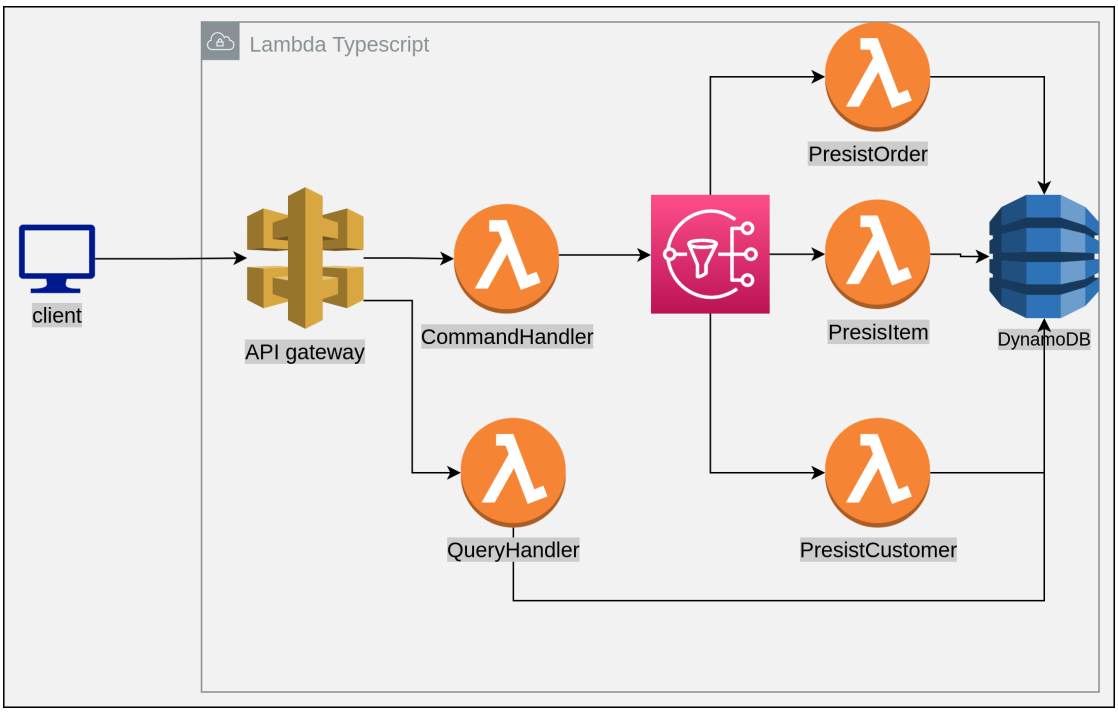


Figure 5.1.: An overview of the architecture of the serverless-resturant application in the latest commit.

5.2. Performance Metrics

As mentioned in the previous sections, the goal of the serverless regression framework is to measure the performance of any target serverless application through the different changes over time. In order to achieve this, we have to first define the performance metrics we intend to collect. Table 5.1 defines a set of metrics, which we refer to as

Metric	Type	Description
http_response_time	user-centric	Total time to receive a response for a given request.
Replicas	platform-centric	Number of total lambda instances.
Memory		Maximum Memory required to run a function.

Table 5.1.: A list of the metrics used, their different types and description

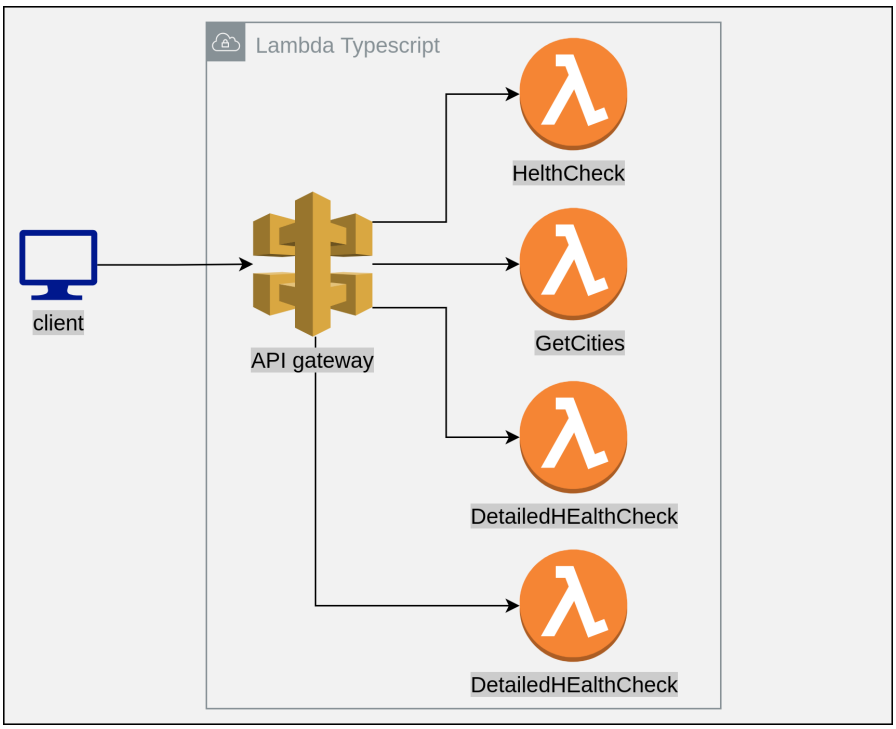


Figure 5.2.: An overview of the lambda typescript application architecture.

platform-centric metrics. To define this set of performance metrics, we conceptualize serverless applications as code snippets that request computing resources on demand, which are provisioned by the serverless platform provider [16]. We focus on the different metrics that measure the change in demands for the different computing resources, such as memory usage, CPU usage, or the time it takes to execute the function. This set of metrics is referred to as platform-centric metrics, as they are specific to the platform of the serverless functions (e.g., AWS).

We also define a different set of performance metrics in Table 5.1, which we refer to as user-centric metrics. This set of metrics is conceptualized from the perspective of how a client, or a user, interacts with a service as a black box. The faster and more frequently a client gets an expected response, the better the service is. The HTTP response time is an excellent example of this type of metric. This set of metrics is referred to as user-centric metrics, as they refer to a set of metrics that measures performance from the perspective of a user or a client acting on a user's behalf.

5.3. Testing process

In this section, we discuss how **RegX** is used to collect the aforementioned set of metrics from the set of application defined in the previous sections. We achieved this by choosing *n* different commits for each application, where n was set to 5. **RegX** is then run with a set of different load testing configurations for each commit. This allows us to test the application at different points in time throughout its development cycle. The set of commits is chosen such that they are as equally apart from each other as possible to ensure enough changes are captured.

The same set of configuration for regression testing is used when running **RegX** to test the different results. To achieve this, scripts from *k6-fdn-load-generator*[3] were used after various contributions were made to the repository. The *k6-fdn-load-generator* can be used to generate load test configuration, using k6, for different kinds of HTTP applications.

After going through the git history and choosing the appropriate commits, k6 scripts tests are created with the relevant HTTP request methods, payloads and the aforementioned load configuration. The results are then sent to influx DB and exported into CSV files, which are used to create various bar and line graphs, which will be discussed in the next chapter. To obtain further insight on the metric data collected, different kinds of aggregation, such as average, 90th percentile, min and max, are calculated and plotted for each trace per each commit. GitLab's *CI/CD* (§2.4.2) is used to ease the process of deploying each commit from the set of chosen commits, as well as destroying them automatically when the tests are done to prepare for the next set of load tests.

5.3.1. Traces

As previously mentioned in (§5.3), different scripts from k6-fdn-load-generator are used to configure the load tests. The configuration is based on the following Table 5.2, which includes the page visit count for different popular Wikipedia articles (also known as traces). These traces are good examples of real-world interaction between users and services. This data was obtained from Kaggle, the world's largest data science community [34]. To properly simulate real world user behavior and test the target applications mentioned earlier, a diverse set across the different traces was used. In this thesis, traces, 2, 4 and 5 were chosen since they represent diverse traffic with different user trends.

[3]https://github.com/Function-Delivery-Network/k6-serverless-load-generator

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	338	280	261	300
Interstellar_de	6	5	2	8

Table 5.2.: A table showcasing a small sample of the different traces used to generate virtual users. This was chosen as it represents real life user interaction with different web pages (i.e., Wikipedia articles in this case)

5.4. Evaluation Questions

We design *RegX* to answer the following questions for an application under evaluation:

- **Q1. Individual Commit Performance**: how does an application perform at the given commit ID ? In this case, we analyze the performance of the application at the given commit ID using all the three traces representing different varieties of user-load patterns. In this regard, we perform analysis from two types of perspectives:

    1. User Centric: here the user-centric metrics (§5.2) are calculated for the application at the given commit ID for all the different traces. This is done to see how the current changes in the application affect the user at different levels of user workloads.

    2. Platform Centric: here the platform-centric metrics (§5.2) are calculated for the application at the given commit ID for all the different traces. This is done to see how the current changes in the application affect the functions' usage on the platform at different levels of user workloads.

- **Q2. Cross Commit Performance**: how does an application performance vary overtime at different commits? In this case, we analyze the performance of the application at the different commits using all the three traces. In this regard, again, we perform analysis from two types of perspectives:

    1. User Centric: here the user-centric metrics (§5.2) are calculated for the application for the different commit IDs for all the different traces. This is done to see how the changes in the application overtime affect the user at different levels of user workloads.

2. Platform Centric: here the platform-centric metrics (§5.2) are calculated for the application at the different commit IDs for all the different traces. This is done to see how the changes in the application overtime affect the functions' usage on the platform at different levels of user workloads.

6. Results

In this chapter, we analyze the different results and data collected from evaluating the set of chosen applications using the evaluation method explained in Chapter 5. For each of the applications, we will analyze both the user and platform-centric metrics (§5.4) for the first commit of the application, followed by the analysis of the same metrics over the five chosen commits for each application. This process is repeated for the three traces (§5.3.1).

6.1. Serverless Restaurant Application

In this section, we will analyze the different user and platform-centric metrics (§5.4) for the serverless Restaurant application (§5.1.1).

6.1.1. Commit Specific Results

In this section, we analyze the user-centric and platform-centric (§5.4) metrics collected for the first commit of the application.

User Centric Results

We start by inspecting how the response time changes for each trace in Figure 6.1. We notice that the lambda functions react differently to different trends of user requests. For example, the response time for running the first trace (Trace 2 Figure 6.1) shows that the application's response time remains relatively low with small spikes, similar to the pattern of requests it receives. On the other hand, we notice three significant spikes in response time for trace 4, which are caused by the sudden spike in requests towards the end of the trace after a consistently low number of requests. This is likely a result of lambda instantiating new lambda instances and the cold start each instance has to

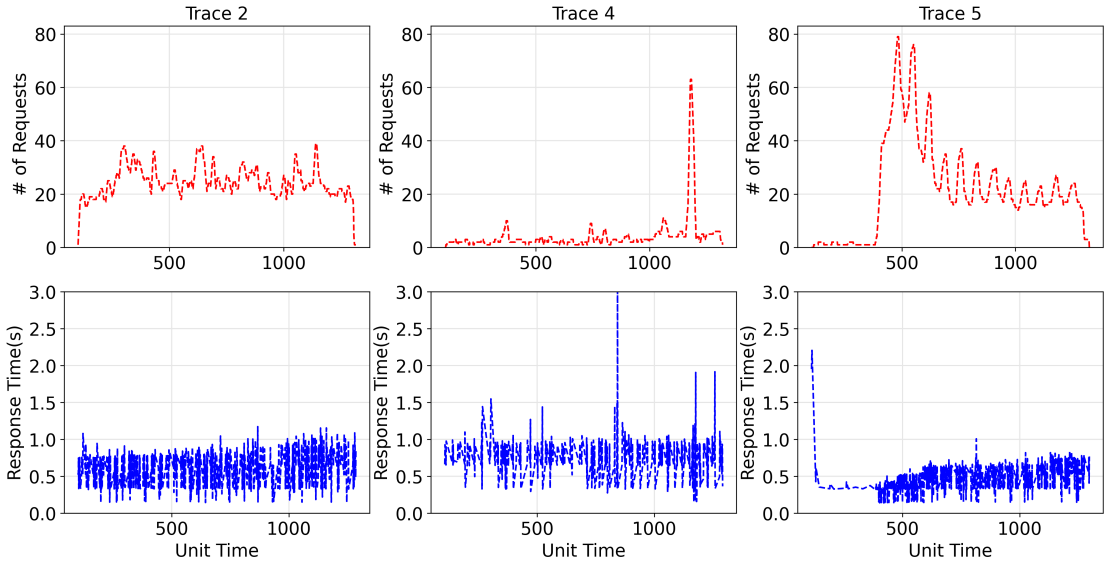


Figure 6.1.: Results of running three different traces (also known as load testing configurations) against **serverless restaurant** application

go through. These new lambda instances are instantiated to accommodate the increase in requests. The last trace (trace 5) shows an initially high response time, which likely results from lambda's cold boot for the first few requests in addition to the cold start of concurrently instantiating more lambda instances to handle the surge in requests. This can be seen at the beginning of the last trace.

To obtain further insights into the collected metrics, we also analyze the average (avg), the 90th percentile P(90), the minimum (min), and maximum (max) aggregates for the *response time* in Figure 6.2. The figure shows that the highest response time recorded for this commit happens at trace 4, which aligns with the spikes seen in Figure 6.1. This confirms that the response time increases significantly after a sudden spike in requests. However, the value of the P(90) suggests that this increase in response time is short-lived and does not significantly impact performance over time. The minimum response time remains similar across the traces, implying that this is the least amount of time required for the application to run, given that enough lambda warm lambda instances exist to accommodate the incoming number of requests and that the request is valid.

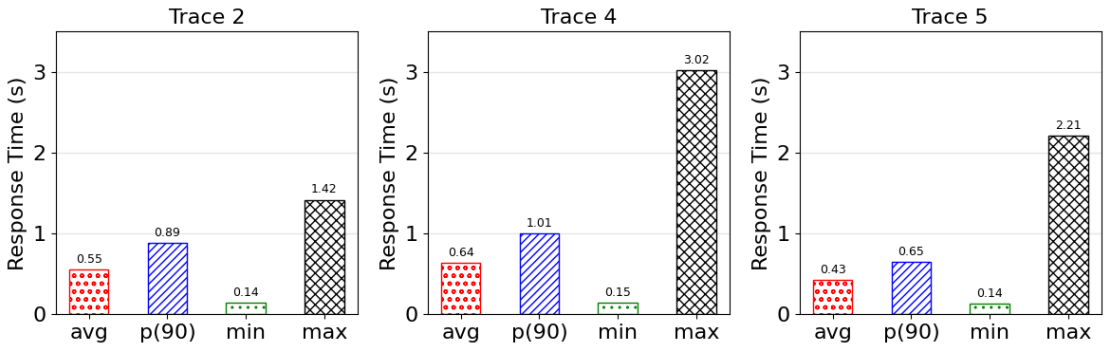


Figure 6.2.: Different aggregates for the response time collected from running the three traces (§5.3.1) against the first commit of the application.

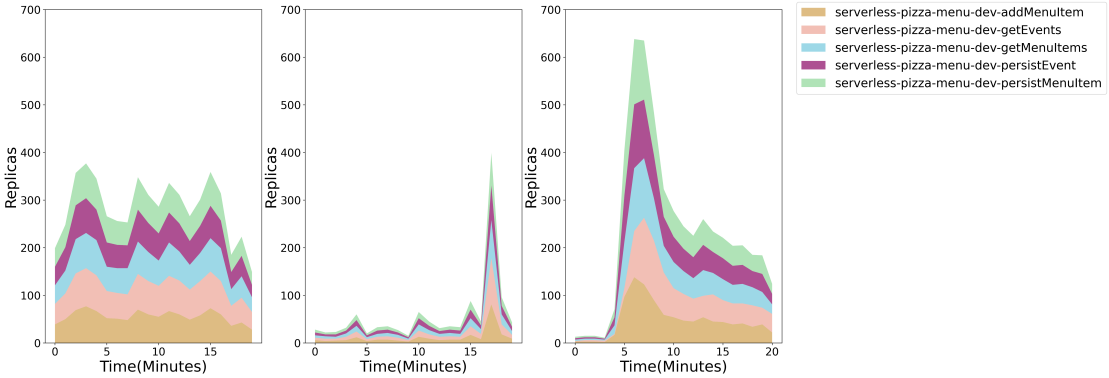


Figure 6.3.: A stacked area graph showing the total number of replicas (also known as concurrent executions) for the first commit of the *serverless-restaurant* for the different traces (§5.3.1)

Platform Centric Results

Figure 6.3 shows a stacked area graph for the replicas of the first commit's functions. Upon inspecting the graph, we notice that the general shapes of the plot are very similar to the shapes of the traces, which can be seen in the first row of Figure 6.1. In fact, after further inspecting the data used to plot the graphs, we noticed a near 1:1 mapping.

Figure 6.4, on the other hand, shows a stacked graph for the memory usage metric for each application across the different traces. Unlike the replicas, the memory usage seems to be relatively consistent across the various traces for this commit. Since the minimum amount of memory allocated for lambda is 128 MB [6], all functions only

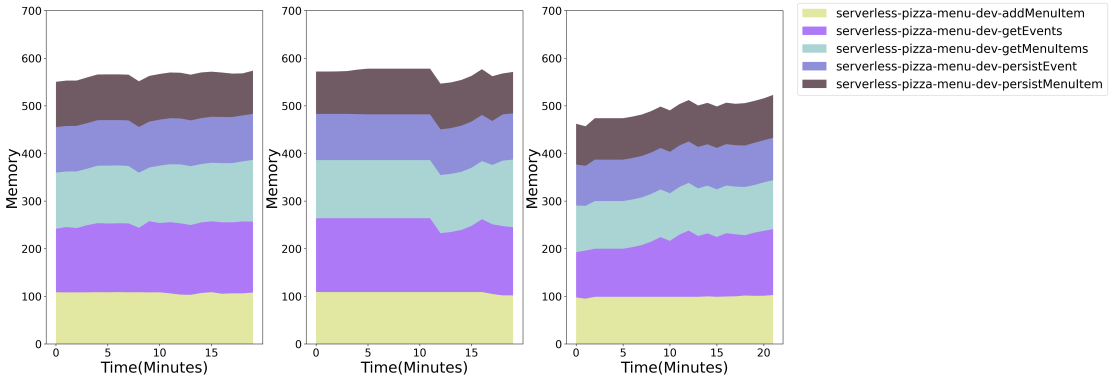


Figure 6.4.: A stacked area graph showing the total memory used in MB for the first commit of the *serverless-restaurant* for the different traces (§5.3.1)

require around this amount, with the total memory usage falling between 500 MB and 750 MB and a somewhat similar area for each function within the application for this commit.

6.1.2. Cross Commit Results

In this section, we analyze the user-centric and platform-centric (§5.4) metrics collected for the five chosen commits of the *serverless restaurant application* (§5.1.1).

User Centric Results

Figure 6.6 shows that the performance time across the five different commits consistently decreases from commits 1 to 5, respectively, until it eventually plateaus. We also notice that trace 4 is the worst performing trace, likely due to the previously mentioned pattern of requests that start consistently low, followed by a sudden spike. These results are further supported by Figure 6.5, which shows the most significant increase in frequency of response time spikes occurring in trace 4. Despite trace 5 having the highest spike in response time, the response time remains relatively low for most of the time it takes to execute trace 5. This is reflected in Figure 6.6 having a lower P(90) value for trace five compared to trace four across the different commits.

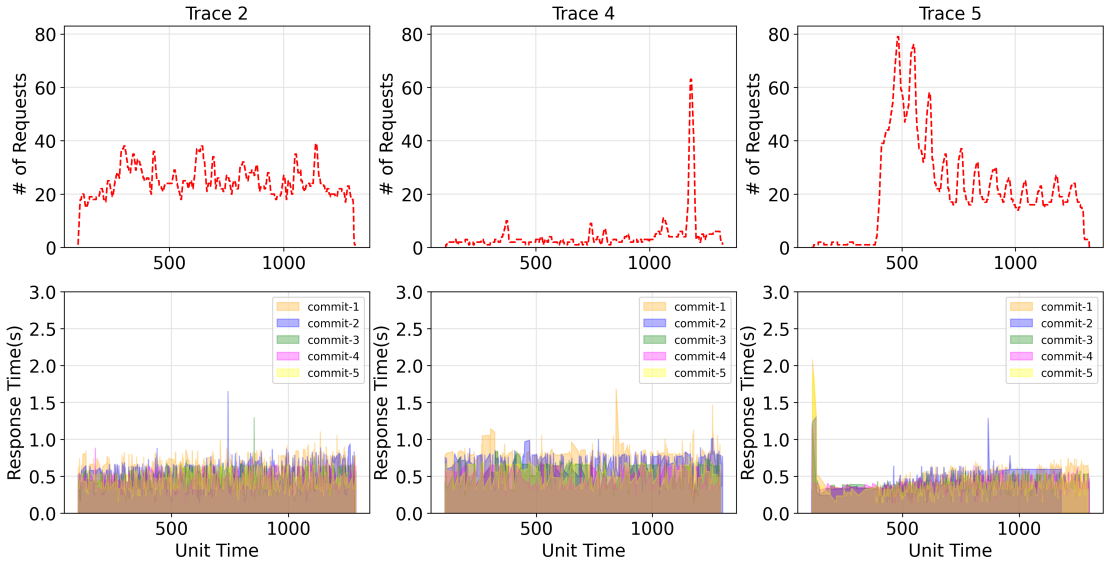


Figure 6.5.: A bar graph showing the P90 response time in seconds for commits 1-5 for the three selected traces from §5.3.1 for the serverless restaurant application.

Platform Centric Results

Figure 6.7 shows that the total number of concurrent executions increases as we move from the first to the fifth commit. This is a direct result of increasing the number of lambda functions across the five commits. The new functions can also be seen in the same figure. The figure also shows that the applications' functions were changed, which introduced complexities in the data collection, yet it was still possible to capture using **RegX**.

On the other hand, Figure 6.8 shows the stacked area graph for the memory usage for each function across the different traces for each commit. Similar to the Figure 6.7, this graph shows an increased total memory usage over the different commits as a result of the increase in the number of lambda functions. However, upon close inspection, we notice that the amount of memory used per function remains relatively similar. This suggests that each function requires around the minimum 128 MB allocated to it to run successfully.

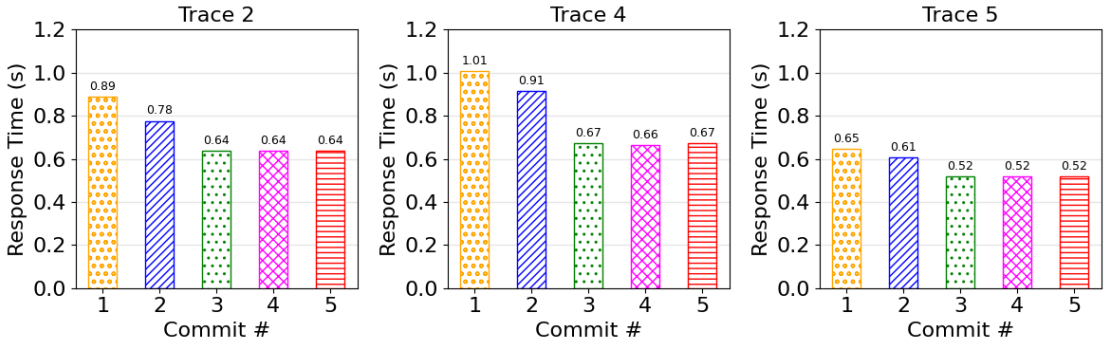


Figure 6.6.: A bar graph showing the response time in seconds for commits 1-5 for the three selected traces from §5.3.1 for the serverless restaurant application.

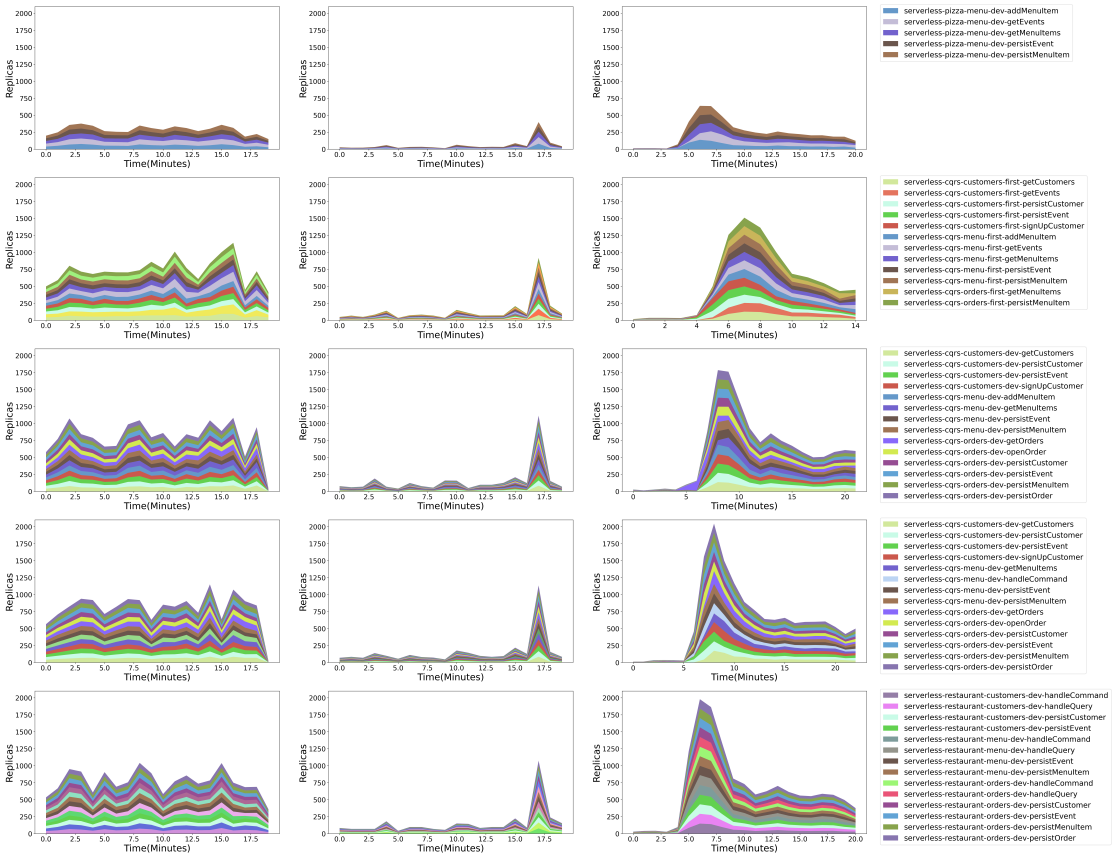


Figure 6.7.: A stacked area graph showing the total number of replicas (also known as concurrent executions) for commits 1-5 of the *serverless-restaurant* application.

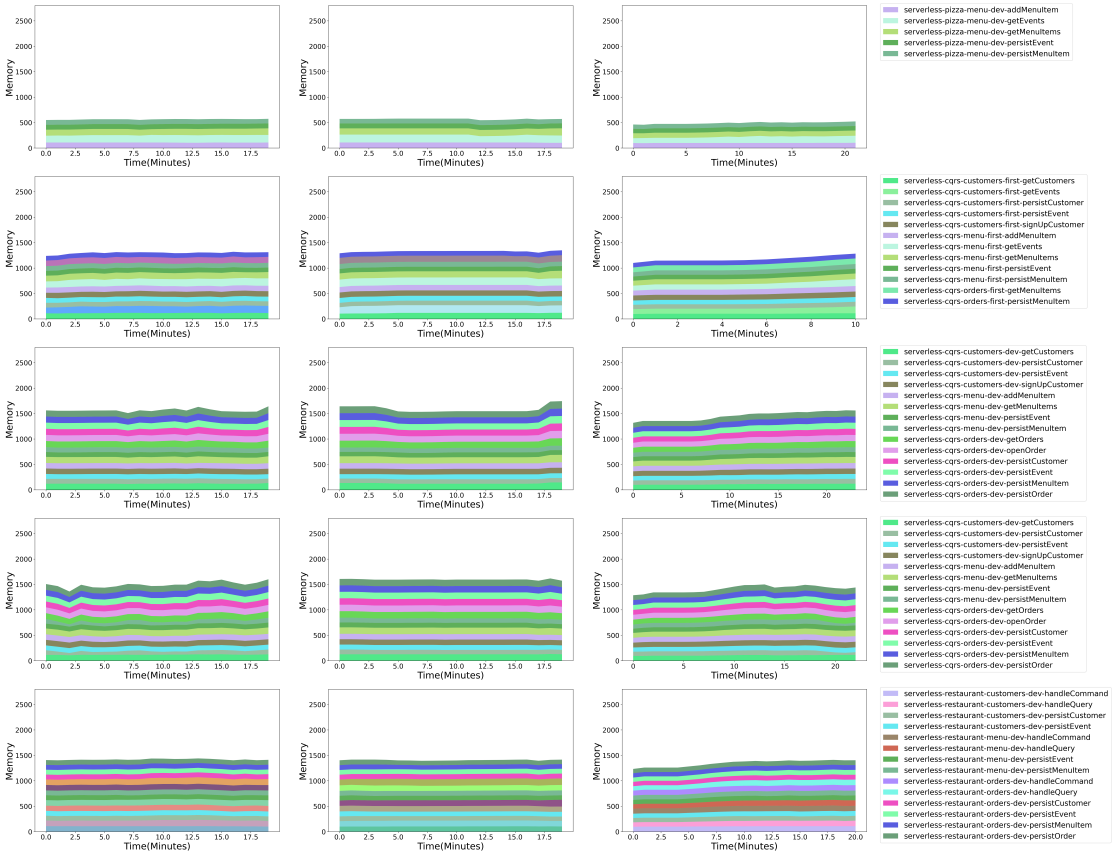


Figure 6.8.: A stacked area graph showing the total memory in MB for commits 1-5 of the *serverless-restaurant* application.

6.2. AWS Lambda Typescript application

This section analyzes the different user and platform-centric metrics (§5.4) for the *Serverless Typescript application* (§5.1.2). The metrics collected are the output of executing the testing process mentioned in §5.3. Similar to the analysis in section 6.1, different results per commit, followed by the results across the different commits for each of the traces (§5.3.1).

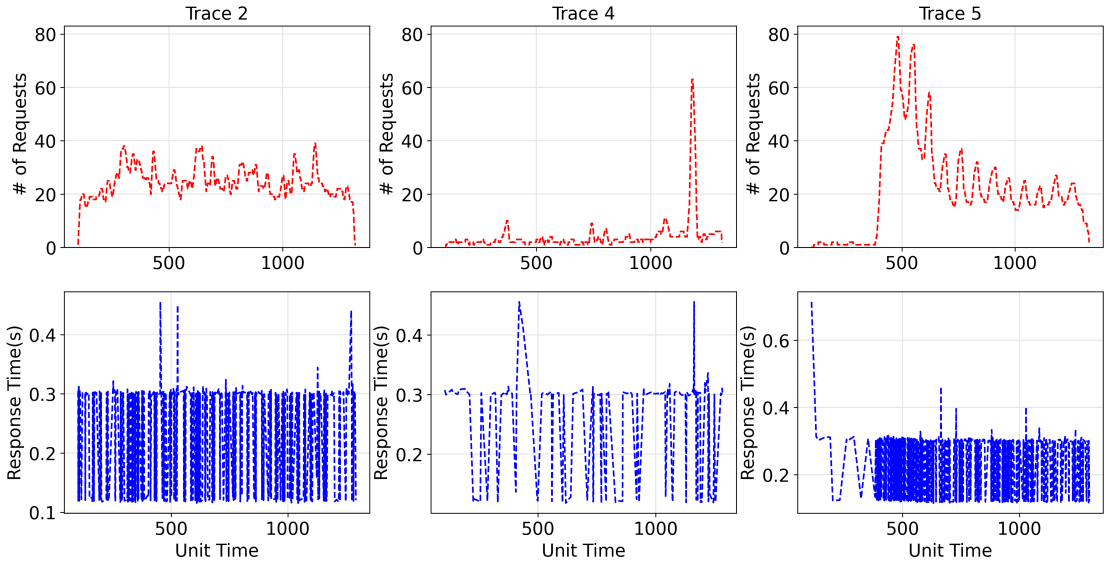


Figure 6.9.: Results of running the three different traces (§5.3.1) for commits 1-5 against **lambda-typescript** application

6.2.1. Commit specific Results

In this section, we analyze the user-centric and platform-centric (§5.4) metrics collected for the first commit of the application.

User Centric Results

We start by analyzing the *HTTP response time* metric for the previously mentioned traces (§5.3.1). Similar to the *serverless restaurant* application (§6.1), we start by analyzing a line graph of the *http response time*. This can be seen in Figure 6.9, which shows a consistently low response time for the different traces. Upon inspecting the state of the application at this commit, we notice that that application only consisted of one lambda function, which will be discussed in the upcoming section. The low response time at this commit is justified given the state of the application. This is further supported by the values of the different aggregates in Figure 6.10, which shows similar values for all aggregates across the different traces.

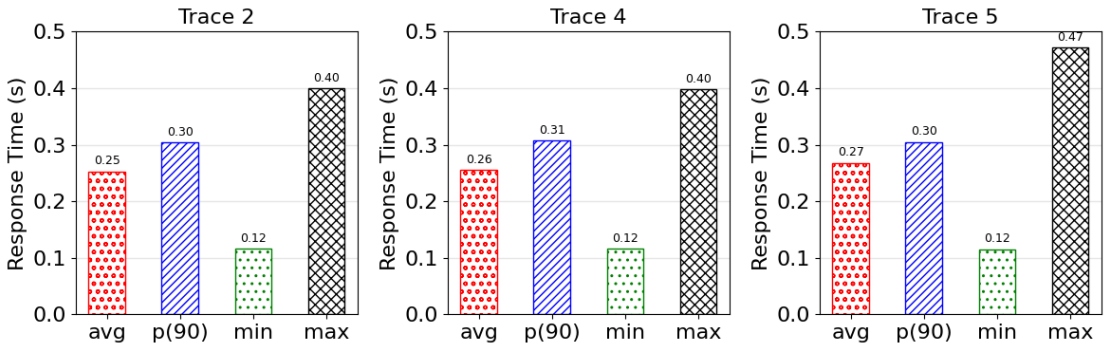


Figure 6.10.: Results of running three different traces (also known as load testing configurations) against **lambda-typescript** application

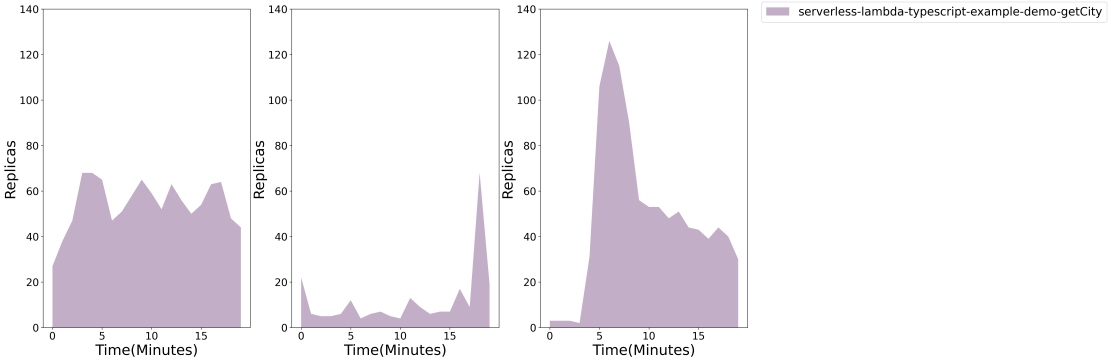


Figure 6.11.: A stacked area graph showing the total number of replicas (also known as concurrent executions) for the first commit of the *lambda-typescript* for the different traces (§5.3.1)

Platform Centric Results

Figure 6.11 and Figure 6.12 confirm that the application starts with a single Lambda function, with the number of replicas in the first figure having a similar pattern to the number of requests in each trace. The second figure shows relatively low memory usage. The number of *replicas* and the *memory used* remains relatively low across the different traces.

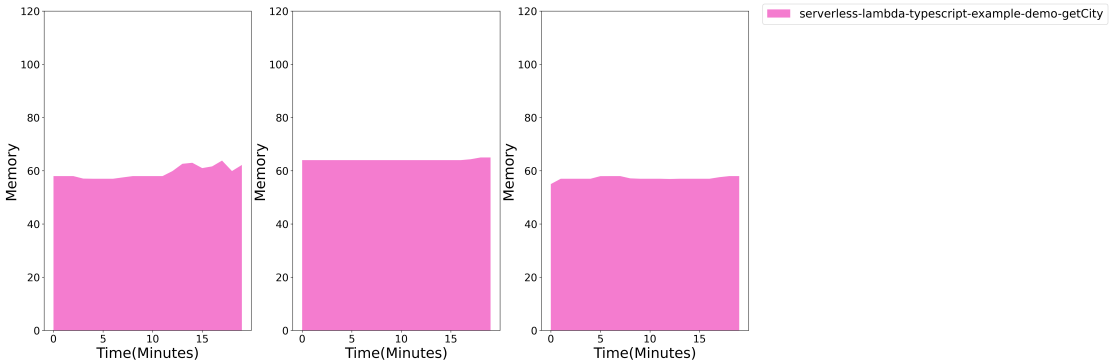


Figure 6.12.: A stacked area graph showing the total Memory used in MB for the first commit of the *lambda-typescript* for the different traces (§5.3.1)

6.2.2. Cross Commit Results

In this section, we analyze the user-centric and platform-centric (§5.4) metrics collected for the five chosen commits of the application.

User Centric Results

Figure 6.14 shows an increase in the *HTTP response time* for all commits after the first commit, except for trace 4. This is also shown in Figure 6.13, which shows consistently higher values for all commits than the first commit.

Platform Centric Results

Figure 6.15 shows that the number of replicas increases significantly after the first commit and remains similar from the second to the fifth commit while retaining the similarity in shape with the different trace line graphs, which can be seen in the first row of Figure 6.13. Similar to the replicas, the memory usage also increases in the application after the first commit. However, the memory usage and replica count per function remain identical, except for the *serverless-sample-demo-getSwaggerjson* function, which consistently has a higher number of replicas and memory used since it was introduced in the second commit.

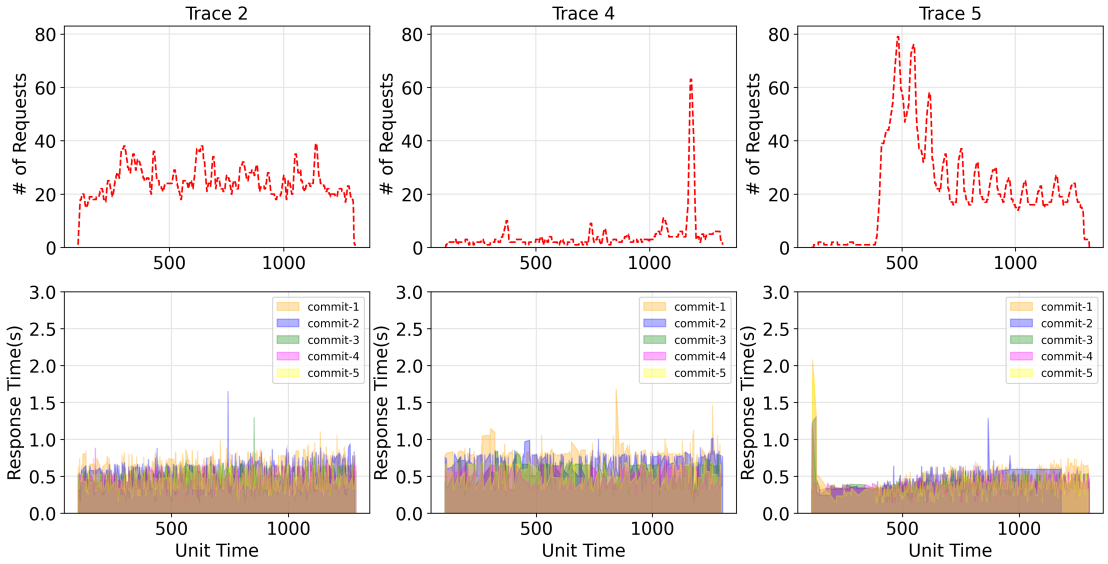


Figure 6.13.: A bar graph showing the P90 response time in seconds throughout the different commits for the three selected traces from §5.3.1 for the serverless-typescript application

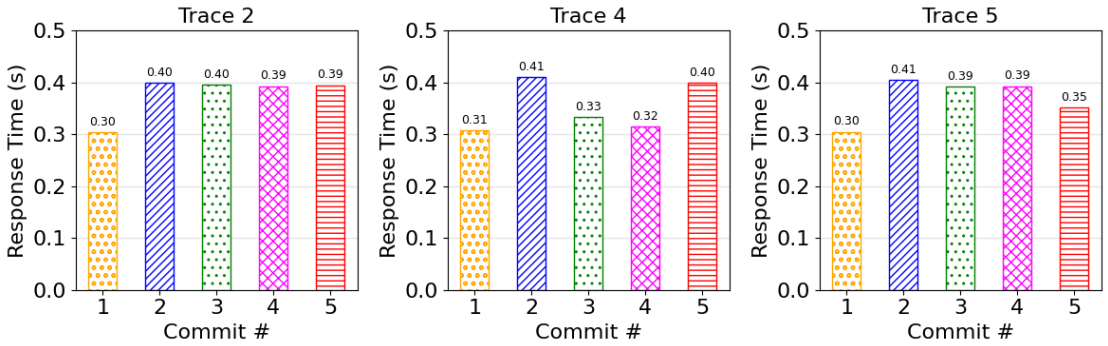


Figure 6.14.: A bar graph showing the response time in seconds throughout the different commits for the three selected traces from §5.3.1 for the serverless-typescript application

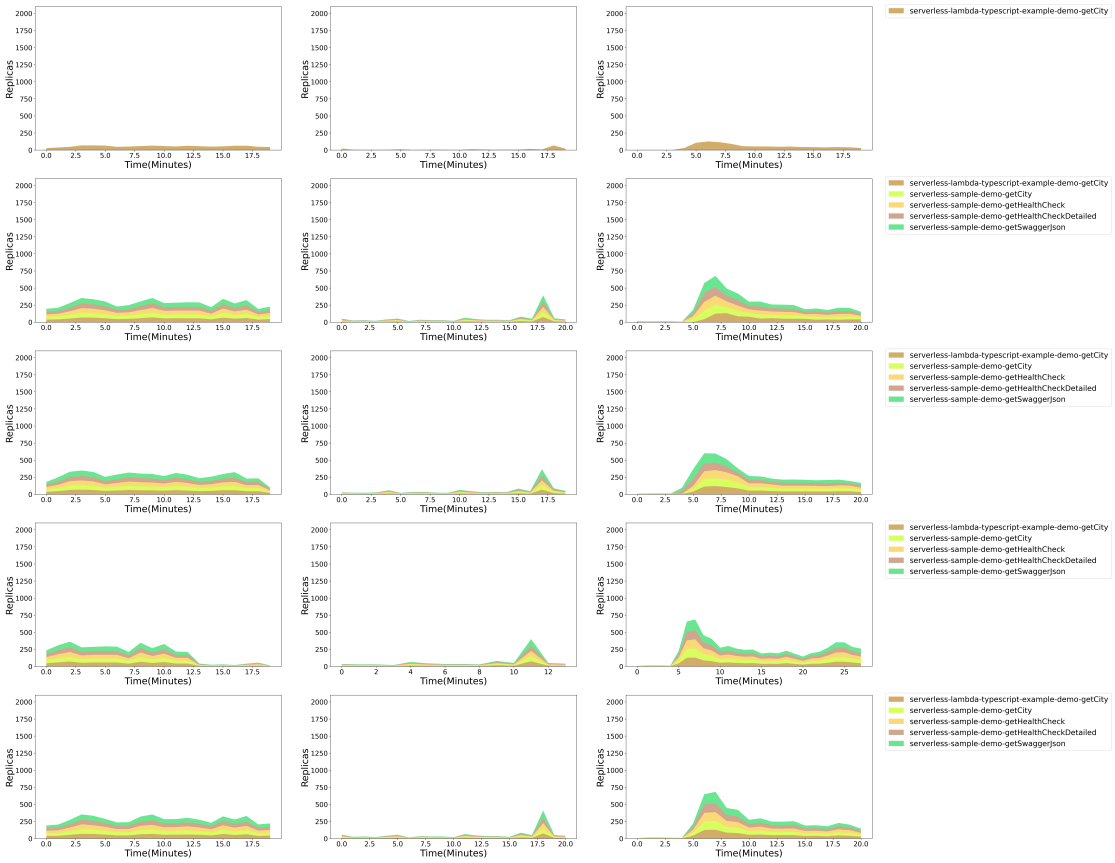


Figure 6.15.: A stacked area graph showing the total memory in MB for commits 1-5 of the *lambda-typescript* for the different traces (§5.3.1)

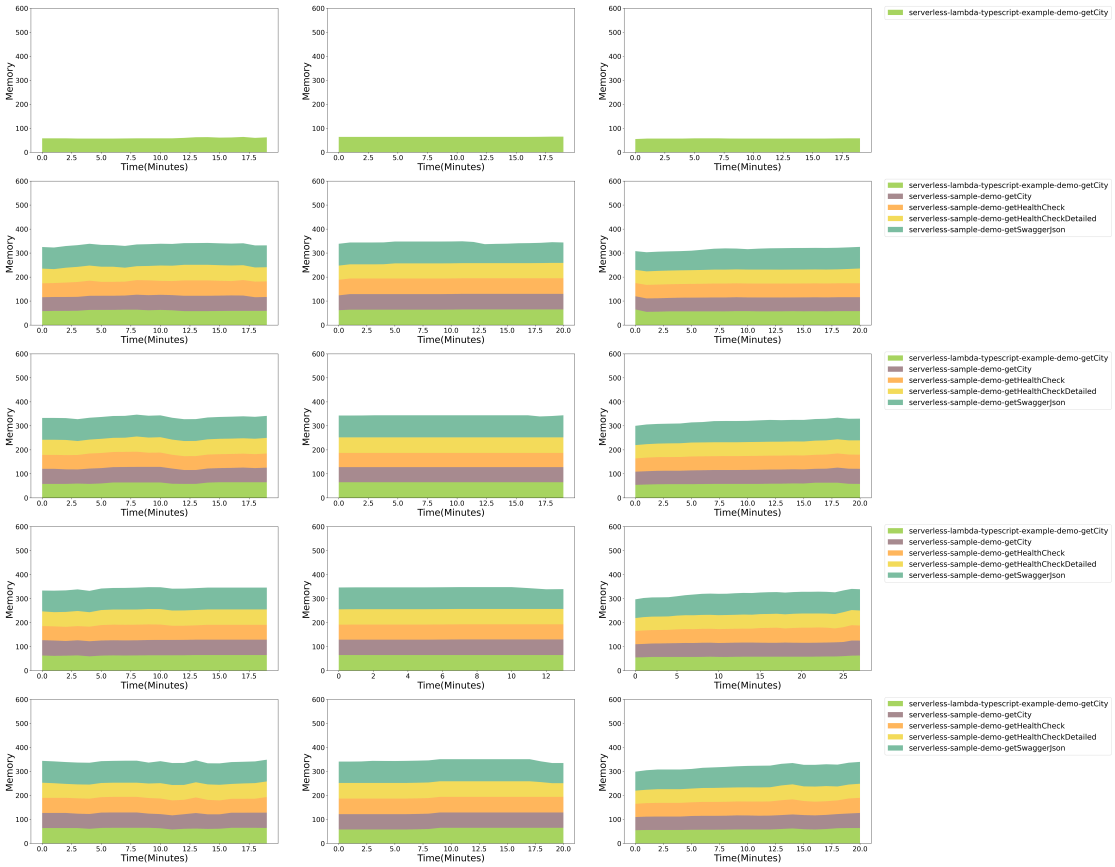


Figure 6.16.: A stacked area graph showing the total number of replicas (also known as concurrent executions) for commits 1-5 of the *lambda-typescript* for the different traces (§5.3.1)

7. Discussion

In this chapter, we discuss the results analyzed in Chapter 6 and what they mean for the performance of each of the applications over the different chosen commits. We also discuss their performance against the three different traces (5.4). We start by discussing the results for the *serverless restaurant application* followed by the *AWS Lambda Typescript application*. For each application, we discuss how the change in total computing resource requirements, represented by the *total memory used* and the *total number of concurrent executions*, impact the application's overall performance.

7.1. Serverless Restaurant Application Performance

The results analyzed in §6.1 show that the application's total resource requirements increase in commits one through five. This increase is a result of the newly introduced function. Despite the increase in total resource requirement for the application, the resource consumption per function remains relatively similar. The results also show consistent improvements in the response time from commits one through five under different loads. These improvements show that the changes introduced by the developers to this application over different commits consistently contributed to better performance at the expense of more resources.

7.2. AWS Lambda Typescript Application Performance

The results analyzed in (§5.1.2) show that the application's resource requirements increased after the first commit and remained relatively similar from commits two through five. Similar to the first application, this is a result of the developers introducing new lambda functions to the applications. While most of the introduced functions require similar resources, the *serverless-sample-demo-getSwaggerJson* function consistently required more resources and had the worst response time. The response time of

this application also seemed to degrade for all commits after the first commit. The response fluctuated in commits two through five but remained worse than the first. The application ended up consuming more resources and degrading in performance.

8. Conclusion

This thesis presented an automatic regression testing framework for serverless application, which was named **RegX** (§4). This framework was used to observe the performance of a set of applications over different load testing configurations.

From the results and discussions in the previous sections, we can conclude that the performance of serverless applications highly depends on the developers' implementation and the changes they introduce with each commit. We observe this in the *Serverless Restaurant Application* (§5.1.1), where the changes introduced consistently resulted in higher performance at the expense of more computing resources. On the other hand, the changes introduced for the *Lambda Typescript Application* resulted in higher computing resources and degradation in performance. Based on these results, we also conclude that *RegX* (§4), the automatic, serverless regression framework proposed in this thesis, can be used to help developers avoid degradation in performance and keep track of resource consumption, both on an application level and a function level. *RegX* achieves this by annotating each change in performance with the commit ID of the commit that introduced this change. The users can then decide if the performance is acceptable if it falls within their SLO by setting a certain threshold to be notified when the performance goes below it.

8.1. Future Work

The **RegX** framework (§4) we developed in this thesis is an initial implementation of an automatic, serverless regression testing framework. There are many possible improvements and upgrades that could be made to the current implementation of **RegX**, which will be discussed in the following sections.

8.1.1. Heterogeneous Platform Support

While **RegX** does include different data collectors for different FaaS platforms, our current implementation only uses the *AWS Data Collector*. This should allow for interesting data collection and comparisons in cases where the same application is deployed on different FaaS providers. It should also allow for capturing performance degradation across different FaaS providers.

8.1.2. Minimum Accepted Performance Threshold

Current static code analysis tools, such as sonarqube [1], can be used to prevent a CI/CD pipeline from passing if the test coverage for a given code base drops below a specific threshold. The **RegX** framework could greatly benefit from this feature. This would ensure that the performance does not go below a given SLO by preventing inefficient code from making it to the code base in the first place. The current implementation of **RegX** only allows the user to inspect the performance, find the commit responsible for a specific change in performance, and set up notifications through *Grafana* (§2.6.2).

8.1.3. Cross CI/CD platform Support

While the current version of **RegX** uses GitLab's CI (§2.4.1) tool, it can also work with any tool that provides similar services, such as *Travis CI*[2] and GitHub actions[3]. This should make **RegX** more robust and easier to deploy on different platforms, making it platform-agnostic.

[1] https://www.sonarqube.org/
[2] https://www.travis-ci.com/
[3] https://github.com/features/actions

A. Commit specific plots

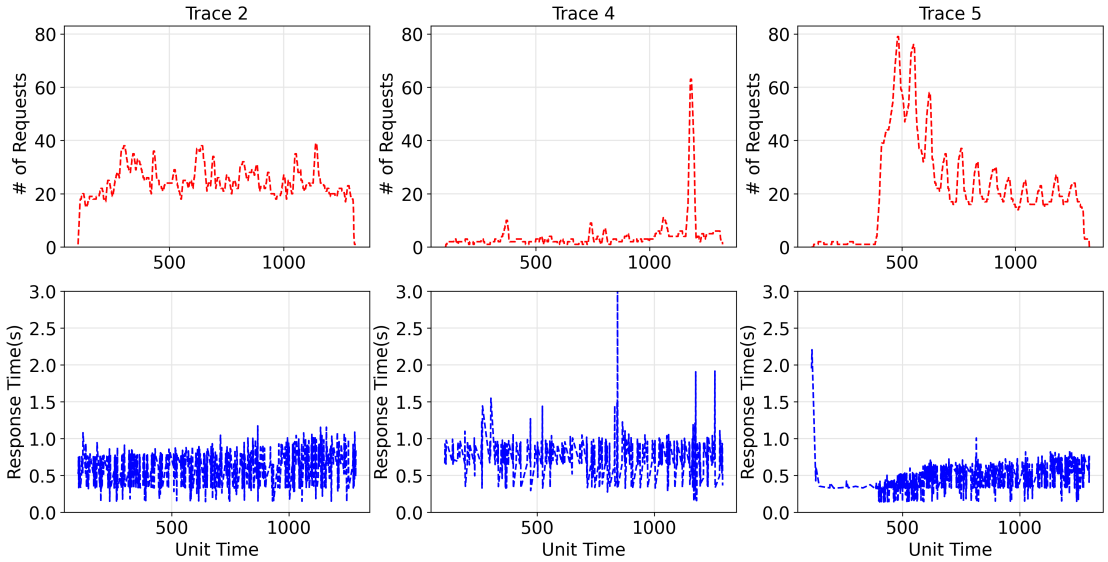


Figure A.1.: A table of images showing the performance of the first commit against the three different traces (§5.3.1)

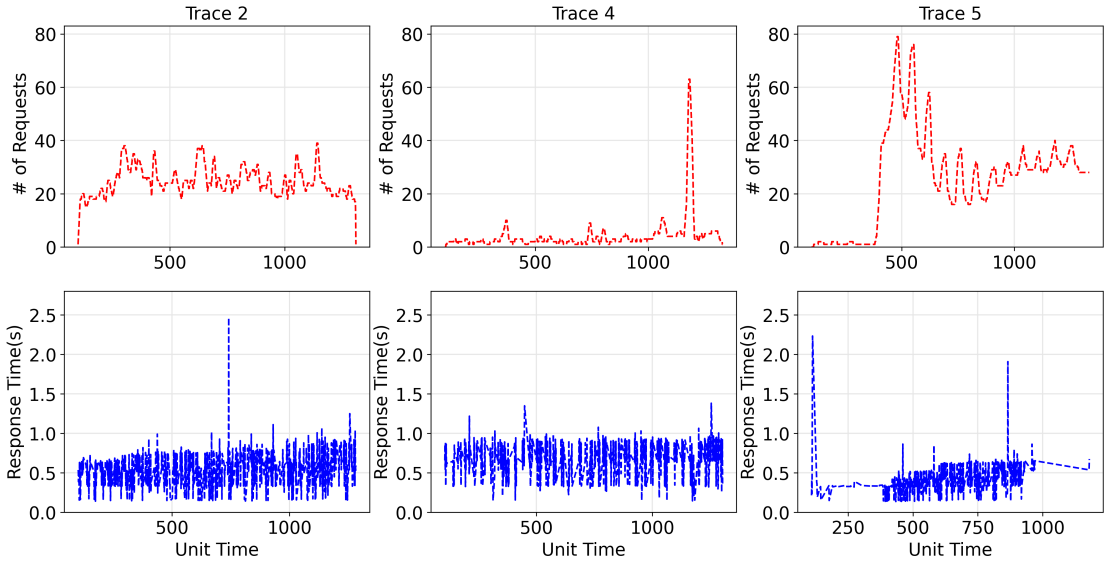


Figure A.2.: A table of images showing the performance of the second commit against the three different traces (§5.3.1)

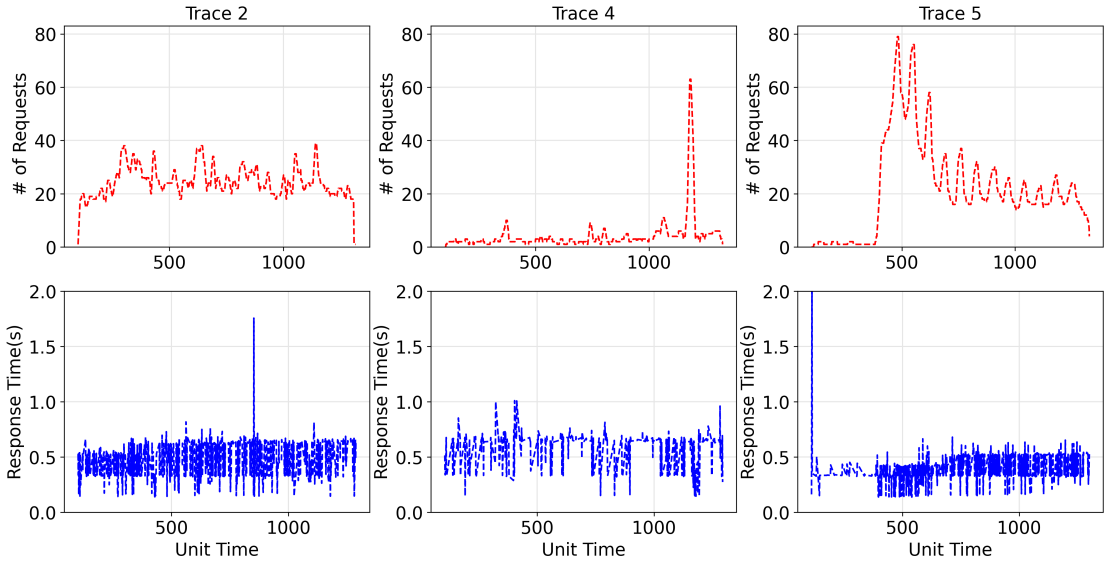


Figure A.3.: A table of images showing the performance of the third commit against the three different traces (§5.3.1)

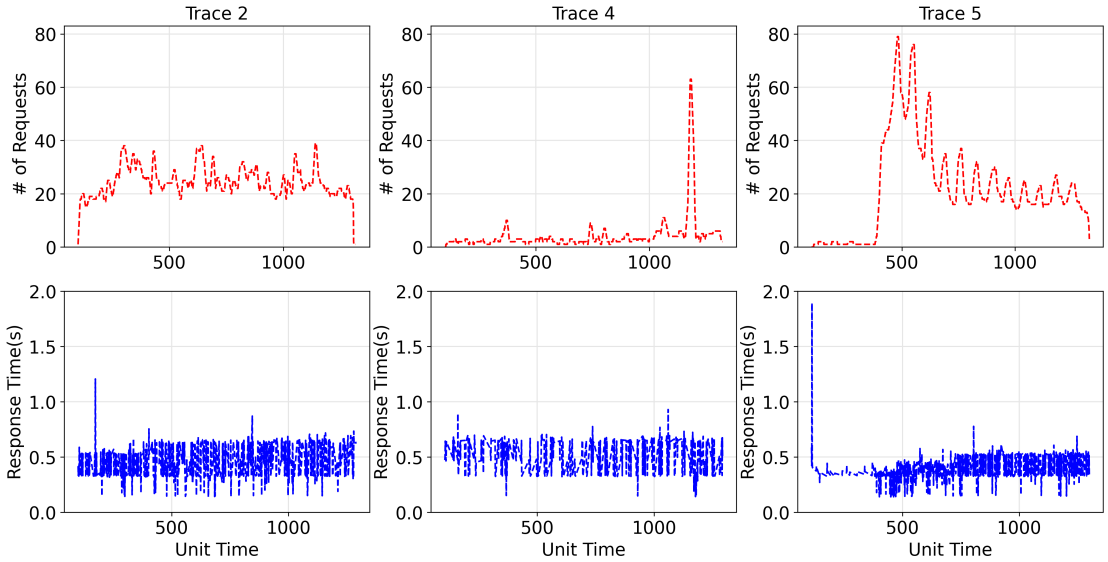


Figure A.4.: A table of images showing the performance of the fourth commit against the three different traces (§5.3.1)

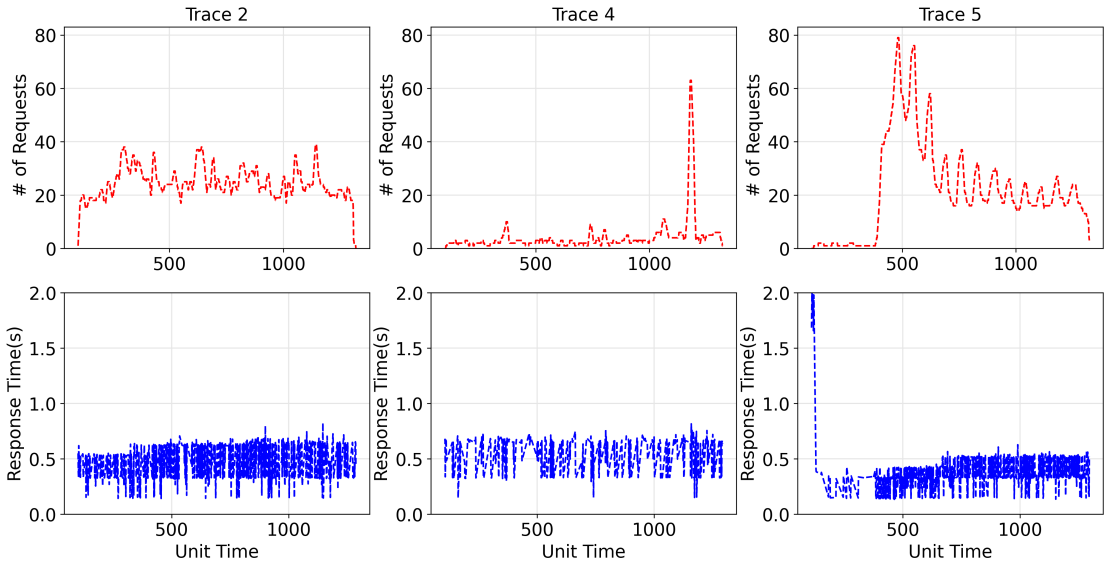


Figure A.5.: A table of images showing the performance of the fifth commit against the three different traces (§5.3.1)

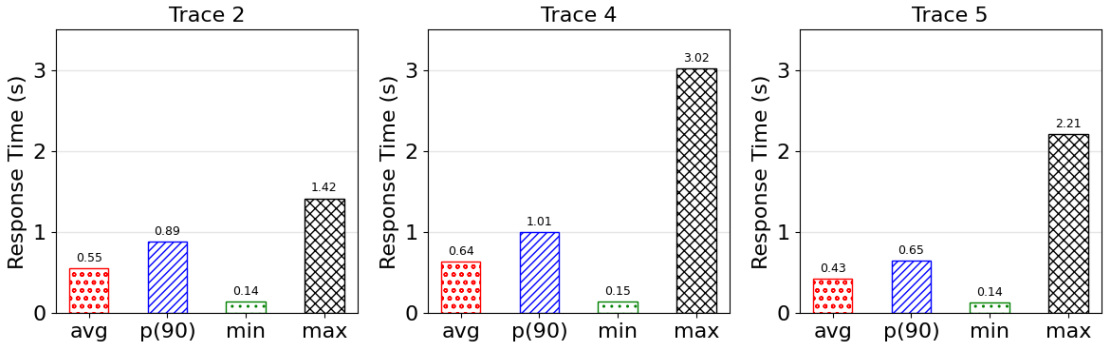


Figure A.6.: A bar graph comparing the average, p(90), minimum and maximum aggregate for the response time of the first commit

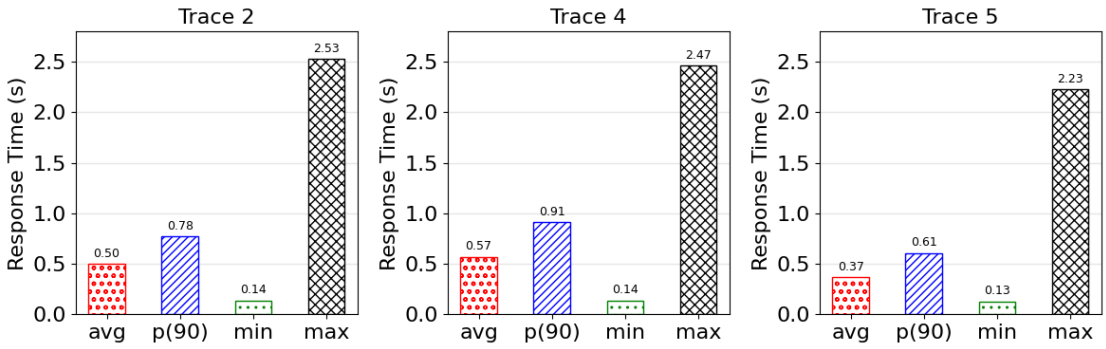


Figure A.7.: A bar graph comparing the average, p(90), minimum and maximum aggregate for the response time of the second commit

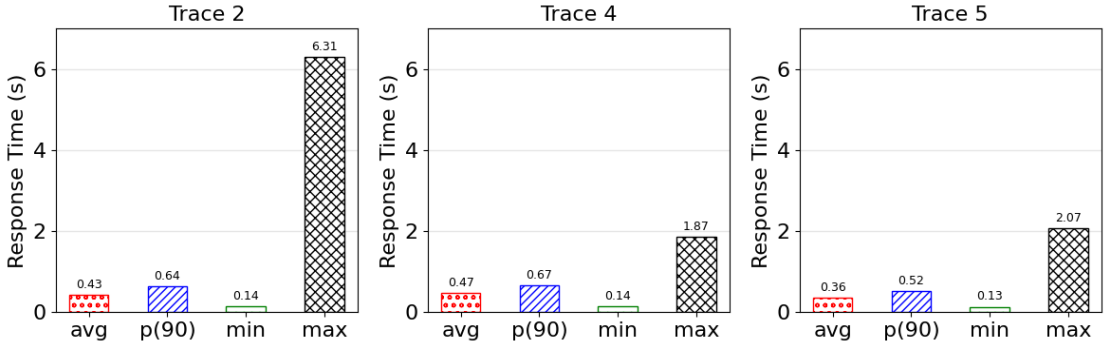


Figure A.8.: A bar graph comparing the average, p(90), minimum and maximum aggregate for the response time of the third commit

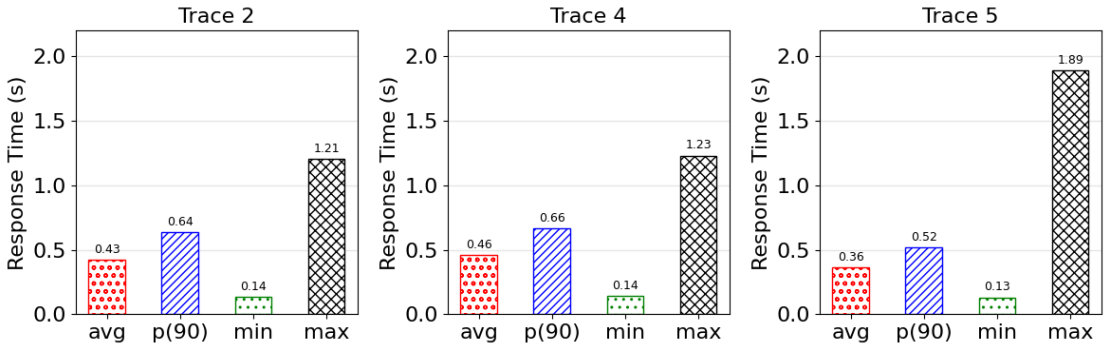


Figure A.9.: A bar graph comparing the average, p(90), minimum and maximum aggregate for the response time of the fourth commit

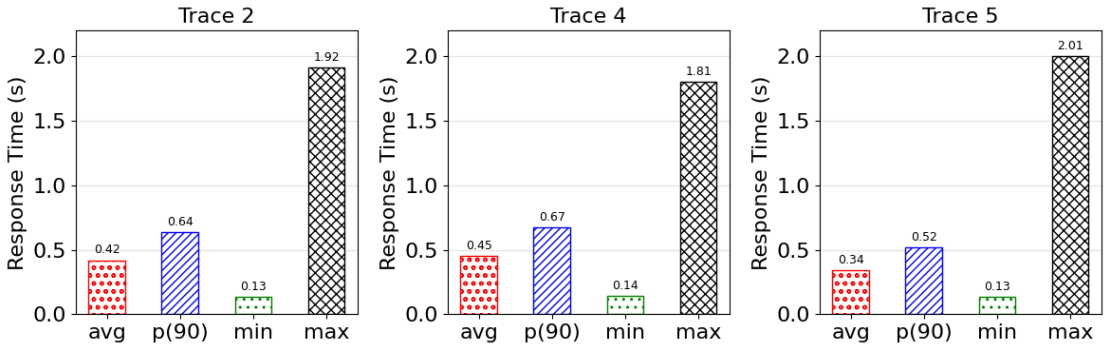


Figure A.10.: A bar graph comparing the average, p(90), minimum and maximum aggregate for the response time of the fifth commit

List of Figures

List of Tables

List of Listings

Bibliography

[1] AWS. *AWS Cloud Development Kit*. https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/aws-cdk.html.

[2] AWS. *AWS SDK for Python (Boto3) Documentation*. https://docs.aws.amazon.com/pythonsdk/.

[3] AWS. *Building Applications with Serverless Architectures*. https://aws.amazon.com/lambda/serverless-architectures-learn-more/. 2021.

[4] AWS. *DevOps Model Defined*. https://aws.amazon.com/devops/what-is-devops/. 2022.

[5] AWS. *Infrastructure as Code*. https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/infrastructure-as-code.html.

[6] AWS. *Memory and computing power*. https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[7] AWS. *Transforming objects with S3 Object Lambda*. https://docs.aws.amazon.com/AmazonS3/latest/userguide/transforming-objects.html.

[8] AWS. *What is Amazon API Gateway?* https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html.

[9] AWS. *What is Amazon CloudWatch?* https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html.

[10] AWS. *What Is Amazon DynamoDB?* https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. 2022.

[11] AWS. *What is Amazon S3?* https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html.

[12] AWS. *What Is AWS?* https://aws.amazon.com/what-is-aws/. 2022.

[13] AWS. *What is IAM?* https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html. 2022.

[14] AWS. *What Is Lambda?* https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. 2022.

[15] AWS. *What Is SNS*. https://docs.aws.amazon.com/sns/latest/dg/welcome.html. 2022.

[16] M. Chadha, A. Jindal, and M. 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.

[17] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, and W. Lloyd. "The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software." In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC'20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 67–72. ISBN: 9781450382045. DOI: 10.1145/3429880.3430103.

[18] T. Dillon, C. Wu, and E. Chang. "Cloud Computing: Issues and Challenges." In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. 2010, pp. 27–33. DOI: 10.1109/AINA.2010.187.

[19] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. "DevOps." In: *IEEE Software* 33.3 (2016), pp. 94–100. DOI: 10.1109/MS.2016.68.

[20] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. "Status of serverless computing and function-as-a-service (faas) in industry and research." In: *arXiv preprint arXiv:1708.08028* (2017).

[21] A. Fuerst and P. Sharma. "FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 386–400. ISBN: 9781450383172. DOI: 10.1145/3445814.3446757.

[22] g3force. *k6 Load Testing Results*. https://grafana.com/grafana/dashboards/15786.

[23] Garima and S. Rani. "Review on time series databases and recent research trends in Time Series Mining." In: *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*. 2014, pp. 109–115. DOI: 10.1109/CONFLUENCE.2014.6949290.

[24] GitLab. *Get started with GitLab CI/CD*. https://docs.gitlab.com/ee/ci/quick_start/.

[25] Gitlab. *GitLab*. https://about.gitlab.com/.

[26] Gitlab. *GitLab CI/CD*. https://docs.gitlab.com/ee/ci/.

[27] M. GroSSmann, C. Ioannidis, and D. T. Le. "Applicability of Serverless Computing in Fog Computing Environments for IoT Scenarios." In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '19 Companion. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 29–34. ISBN: 9781450370448. DOI: 10.1145/3368235.3368834.

[28] InfluxDB. *Time series database (TSDB) explained*. https://www.influxdata.com/time-series-database/.

[29] V. Ivanov and K. Smolander. "Implementation of a DevOps Pipeline for Serverless Applications." In: *Product-Focused Software Process Improvement*. Ed. by M. Kuhrmann, K. Schneider, D. Pfahl, S. Amasaki, M. Ciolkowski, R. Hebig, P. Tell, J. Klünder, and S. Küpper. Cham: Springer International Publishing, 2018, pp. 48–64. ISBN: 978-3-030-03673-7.

[30] A. Janes and B. Russo. "Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices." In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, pp. 163–168. DOI: 10.1109/ISSREW.2019.00067.

[31] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen. "Function delivery network: Extending serverless computing for heterogeneous platforms." In: *Software: Practice and Experience* (Mar. 2021). DOI: 10.1002/spe.2966.

[32] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen. "Function delivery network: Extending serverless computing for heterogeneous platforms." In: *Software: Practice and Experience* 51.9 (2021), pp. 1936–1963.

[33] K6. *K6 Glossary: Virtual users*. https://k6.io/docs/misc/glossary/#virtual-users. 2022.

[34] Kaggle. *Wikipedia Traffic Data Exploration*. https://www.kaggle.com/code/muonneutrino/wikipedia-traffic-data-exploration/notebook.

[35] D. Khatri, S. K. Khatri, and D. Mishra. "Potential Bottleneck and Measuring Performance of Serverless Computing: A Literature Study." In: *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2020, pp. 161–164. DOI: 10.1109/ICRITO48877.2020.9197837.

[36] P. Kruchten, R. L. Nord, and I. Ozkaya. "Technical Debt: From Metaphor to Theory and Practice." In: *IEEE Software* 29.6 (2012), pp. 18–21. DOI: 10.1109/MS.2012.167.

[37] A. Li, X. Yang, S. Kandula, and M. Zhang. "CloudCmp: Comparing Public Cloud Providers." In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: Association for Computing Machinery, 2010, pp. 1–14. ISBN: 9781450304832. DOI: 10.1145/1879141.1879143.

[38] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. "Agile Cold Starts for Scalable Serverless." In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019.

[39] K. Morris. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.

[40] k. D. A. Nicole van der Hoeven. *Comparing k6 and JMeter for load testing*. https://k6.io/blog/k6-vs-jmeter/. 2021.

[41] k. D. A. Nicole van der Hoeven. *Comparing k6 and JMeter for load testing*. https://k6.io/blog/k6-vs-jmeter/. 2021.

[42] R. Pagliuca. "Extending Serverless Computing for Heterogeneous Edge Devices." Masterarbeit. Technische Universität München, 2021.

[43] R. A. P. Rajan. "Serverless Architecture - A Revolution in Cloud Computing." In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. 2018, pp. 88–93. DOI: 10.1109/ICoAC44903.2018.8939081.

[44] Redhat. *What does an API gateway do?* https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do.

[45] Redhat. *What is CI/CD?* https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[46] V. Sarcar. "AntiPatterns: Avoid the Common Mistakes." In: *Java Design Patterns: A Hands-On Experience with Real-World Examples*. Berkeley, CA: Apress, 2019, pp. 467–474. ISBN: 978-1-4842-4078-6. DOI: 10.1007/978-1-4842-4078-6_28.

[47] J. Scheuner and P. Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review." In: *Journal of Systems and Software* 170 (2020), p. 110708. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110708.

[48] S. Sivasubramanian. "Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service." In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 729–730. ISBN: 9781450312479. DOI: 10.1145/2213836.2213945.

[49] W. Tan, L. Fong, and Y. Liu. "Effectiveness Assessment of Solid-State Drive Used in Big Data Services." In: *2014 IEEE International Conference on Web Services*. 2014, pp. 393–400. DOI: 10.1109/ICWS.2014.63.

[50] M. turan. *How to add k6 load test to Gitlab CI pipeline with AWS Lambda Containers*. https://medium.com/modanisa-engineering/how-to-add-k6-load-test-to-gitlab-ci-pipeline-with-aws-lambda-containers-2096278d5711. 2022.

[51] R. Xu, W. Jin, and D. Kim. "Microservice Security Agent Based On API Gateway in Edge Computing." In: *Sensors* 19.22 (2019). ISSN: 1424-8220. DOI: 10.3390/s19224905.

[52] G. Young. "Cqrs documents by greg young." In: *Young* 56 (2010).