TUM School of Computation, Information and Technology
Technische Universität München

Platform Orchestration and Resource Provisioning in Edge-Cloud Infrastructures

Vittorio Cozzolino

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

Doktors der Naturwissenshaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr. Pramod Bathotia

Prüfer*innen der Dissertation:

1. Prof. Dr.-Ing. Jörg Ott
2. Prof. Dr. Sasu Tarkoma,
   University of Helsinki
3. Associate Professor Aaron Ding, Ph.D.,
   Delft University of Technology

Die Dissertation wurde am 05.09.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 24.05.2023 angenommen.

I dedicate this thesis to my family, friends,
and everyone who has supported me along the way.

To my parents, thank you for always believing in me and supporting my dreams.
Your love and encouragement have meant the world to me.

To my sister, thank you for always being there for me, no matter what.
You are my best friends and I am so grateful to have you in my life.

To my colleagues and friends, thank you for being my sounding board, my cheerleaders,
and my shoulder to cry on. I couldn't have done this without you.

To my mentors, thank you for sharing your knowledge and wisdom with me. You have
helped me to grow as a scholar and a person.

To the faculty at the Technical University of Munich, thank you for providing me with
the opportunity to learn and grow. I am so grateful for your support.

And finally, to everyone else who has supported me along the way, thank you.
I am so grateful for your love and encouragement.

Abstract

Cloud computing has hugely impacted over the last few decades the Information Technology (IT) industry by offering virtually unlimited storage and processing capabilities at low-cost enabling new computing models where virtualized resources are leased on-demand. While cloud computing has been dominating the Internet services delivery model for the past thirty years, new technologies and applications for mobile computing and the Internet of Things (IoT) have been pushing towards the decentralization of compute resources. This new trend has been fueled by the requirements of such services including: (*i*) reduction of network latency for delay-sensitive application such as augmented reality, (*ii*) data aggregation and sensor fusion required by the proliferation of IoT and smart devices in order to reduce uplink bandwidth utilization and optimize response time in critical situations, and (*iii*) better customization through context-awareness of location-dependent services. Traditional cloud offloading techniques are ill-suited for these scenarios due to the burden of additional network delay intrinsically encountered while accessing or uploading resources to remote datacenters.

Edge computing is a paradigm in which computing and storage resources are placed at the edge of the network, in close proximity to mobile devices or smart sensors. It aims to bring existing cloud services and utilities near end-users to reduce the network load on the cloud and increase the quality of experience in the case of latency-critical applications, among others. Hence, it can help in addressing shortcomings of the cloud by offering localized compute resources and services. However, full integration between edge and cloud infrastructures is still far from reality. In fact, the deployment and management of an edge infrastructure remains a more challenging task when compared to the cloud for multiple reasons such as the presence of resource-constrained devices, scalability, maintenance, management, and security issues. Therefore, the problem we tackle in this thesis is multi-faceted. The challenges we discuss are at the intersection of system design, resource provisioning at scale, and application requirements. Furthermore, it is paramount for edge computing to be compatible with the traditional cloud technologies so that existing cloud-based applications can be ported to and integrated in edge environments with minimal overhead. This would also flatten the learning curve for developers who would be able to naturally make use of the edge-cloud infrastructure.

This thesis identifies critical integration challenges between edge and cloud computing to create a homogeneous infrastructure abstraction in spite of the apparent duality between the two technologies. We focus on three facets of our approach: domain-specific applications, platform orchestration, and resource provisioning. For the first, we look at examples of different classes of applications that would benefit from the edge computing

paradigm. Then, we deep-dive into novel lightweight virtualization technologies which can be leveraged to build a flexible edge-cloud offloading platform. In particular, this thesis contributes a virtualization-dependent orchestration framework based on unikernels supporting distributed tasks execution and stateful service migration. Finally, from the resource provisioning angle, we provide a latency-energy efficient allocation algorithm to provision resources for tasks offloaded by mobile devices to a multi-tier edge-cloud architecture.

Zusammenfassung

Das Cloud Computing hat in den letzten Jahrzehnten einen enormen Einfluss auf die IT-Branche ausgeübt, da es praktisch unbegrenzte Speicher- und Verarbeitungskapazitäten zu niedrigen Kosten bietet und neue Computermodelle ermöglicht, bei denen virtualisierte Ressourcen auf Abruf gemietet werden. Während das Cloud Computing in den letzten dreißig Jahren das Modell für die Bereitstellung von Internetdiensten dominiert hat, drängen neue Technologien und Anwendungen für das mobile Computing und das IoT auf die Dezentralisierung von Rechenressourcen. Dieser neue Trend wurde durch die Anforderungen solcher Dienste angeheizt, darunter: (*i*) Verringerung der Netzwerklatenz für verzögerungsempfindliche Anwendungen wie Augmented Reality, (*ii*) Datenaggregation und Sensorfusion, die durch die Verbreitung von IoT und intelligenten Geräten erforderlich sind, um die Nutzung der Uplink-Bandbreite zu verringern und die Reaktionszeit in kritischen Situationen zu optimieren, und (*iii*) bessere Anpassung durch kontextabhängige Dienste. Herkömmliche Cloud-Offloading-Techniken sind für diese Szenarien aufgrund der zusätzlichen Netzwerkverzögerung, die beim Zugriff auf oder Hochladen von Ressourcen in entfernte Rechenzentren auftritt, schlecht geeignet.

Edge Computing ist ein Paradigma, bei dem Rechen- und Speicherressourcen am Rande des Netzes, in unmittelbarer Nähe zu mobilen Geräten oder intelligenten Sensoren, platziert werden. Es zielt darauf ab, bestehende Cloud-Dienste und -Hilfsmittel in die Nähe der Endnutzer zu bringen, um die Netzbelastung der Cloud zu verringern und die Qualität der Erfahrung bei latenzkritischen Anwendungen zu verbessern. Sie kann also dazu beitragen, die Unzulänglichkeiten der Cloud zu beheben, indem sie lokalisierte Rechenressourcen und Dienste anbietet. Die vollständige Integration von Edge- und Cloud-Infrastrukturen ist jedoch noch weit von der Realität entfernt. In der Tat ist der Einsatz und die Verwaltung einer Edge-Infrastruktur im Vergleich zur Cloud aus mehreren Gründen eine größere Herausforderung, z. B. durch das Vorhandensein ressourcenbeschränkter Geräte, Skalierbarkeit, Wartung, Verwaltung und Sicherheit. Daher ist das Problem, das wir in dieser Arbeit angehen, vielschichtig. Die Herausforderungen, die wir erörtern, liegen an der Schnittstelle von Systemdesign, Ressourcenbelegung in großem Umfang und Anwendungsanforderungen. Darüber hinaus ist es von größter Bedeutung, dass Edge Computing mit den herkömmlichen Cloud-Technologien kompatibel ist, damit bestehende Cloud-basierte Anwendungen mit minimalem Aufwand in Edge-Umgebungen portiert und dort integriert werden können. Dies würde auch die Lernkurve für Entwickler abflachen, die in der Lage wären, die Edge-Cloud-Infrastruktur auf natürliche Weise zu nutzen.

In dieser Arbeit werden kritische Herausforderungen bei der Integration von Edge- und Cloud-Computing identifiziert, um trotz der scheinbaren Dualität der beiden Technologien eine homogene Infrastrukturabstraktion zu schaffen. Wir konzentrieren uns auf drei Facetten unseres Ansatzes: domänenspezifische Anwendungen, Plattform-Orchestrierung und Ressourcenbereitstellung. Für den ersten Aspekt betrachten wir Beispiele verschiedener Anwendungsklassen, die vom Paradigma des Edge Computing profitieren würden. Anschließend befassen wir uns eingehend mit neuartigen, leichtgewichtigen Virtualisierungstechnologien, die für den Aufbau einer flexiblen Edge-Cloud-Offloading-Plattform genutzt werden können. Insbesondere wird in dieser Arbeit ein virtualisierungsabhängiger Orchestrierungsrahmen auf der Basis von Unikerneln vorgestellt, der die Ausführung verteilter Aufgaben und die zustandsabhängige Migration von Diensten unterstützt. Schließlich bieten wir aus dem Blickwinkel der Ressourcenbereitstellung einen latenz- und energieeffizienten Zuweisungsalgorithmus zur Bereitstellung von Ressourcen für Aufgaben, die von mobilen Geräten auf eine mehrstufige Edge-Cloud-Architektur ausgelagert werden.

Contents

List of Figures

List of Tables

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their numeral. All publications are subject to a full peer-review process:

[**1**]. Roberto Morabito, **Vittorio Cozzolino**, Aaron Yi Ding, Nicklas Beijar, and Jörg Ott. "Consolidate IoT Edge Computing with Lightweight Virtualization." *IEEE Network 32*, no. 1, pp. 102-111. 2018.[1]

[**2**]. **Vittorio Cozzolino**, Aaron Yi Ding, and Jörg Ott. "FADES: Fine-grained Edge Offloading with Unikernels." *In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, pp. 36-41. 2017.

[**3**]. **Vittorio Cozzolino**, Oliver Flum, Aaron Yi Ding, and Jörg Ott. "MirageManager: Enabling Stateful Migration for Unikernels." *In Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*, pp. 13-19. 2020.

[**4**]. **Vittorio Cozzolino**, Tonetto Leonardo, Nitinder Mohan, Aaron Yi Ding, and Jörg Ott. "Nimbus: Towards Latency-Energy Efficient Task Offloading for AR Services." *IEEE Transaction on Cloud Computing*, 2022.

[**5**]. **Vittorio Cozzolino**, Oleksii Moroz, and Aaron Yi Ding. "The Virtual Factory: Hologram-Enabled Control and Monitoring of Industrial IoT Devices." *In 2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pp. 120-123. 2018.

[**6**]. **Vittorio Cozzolino**, Aaron Yi Ding, and Jörg Ott. "Edge Chaining Framework for Black Ice Road Fingerprinting." *In Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, pp. 42-47. 2019.

[**7**]. **Vittorio Cozzolino**, Nikolai Schwellnus, Jörg Ott, and Aaron Yi Ding. "UIDS: Unikernel-based Intrusion Detection System for the Internet of Things." *In Workshop on Decentralized IoT Systems and Security (DISS)*, 2020.

[**8**]. **Vittorio Cozzolino**, Aaron Yi Ding, Richard Mortier, and Jörg Ott. "ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment." *In*

[1] Alphabetical ordering was applied due to equal contributions from the first two authors. This is stated also in the original manuscript.

2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), pp. 223-230. 2020.

Author's Contributions

Publication I: Consolidate IoT Edge Computing with Lightweight Virtualization.

The paper idea originated with Roberto Morabito, Aaron Yi Ding, and I. Roberto Morabito and I equally contributed to the content of the article. I contributed to the formulation of the problem and discussed the applicability of a specific form of Lightweight Virtualization to a set of application scenarios. In particular, I was involved in the analysis of how unikernels can be applied to sensor data processing in IoT scenarios and for real-time applications. The discussion section was a shared work between all the authors and I specifically focused on elasticity in service provisioning and application portability.

Publication II: FADES: Fine-grained Edge Offloading with Unikernels.

I came up with the idea for the paper as a foundation for an unikernel-based orchestration system. I have designed, implemented, and evaluated the entire system.

Publication III: MirageManager: Enabling Stateful Migration for Unikernels.

I came up with the idea for the paper as a foundation for the stateful migration of unikernels. I contributed to the system design aspects and research foundation, while Oliver Flum implemented and evaluated the system.

Publication IV: Nimbus: Towards Latency-Energy Efficient Task Offloading for AR Services.

I came up with the idea for the paper as a foundation for task provisioning in edge-cloud infrastructures. I have designed, implemented, and evaluated the entire system. Leonardo Tonetto supported the design phase and offered valuable insights for the data interpretation part. Nitinder Mohan helped by providing a dataset that was crucial for the network benchmarks. Leonardo Tonetto and Nitinder Mohan also supported the writing of the final publication.

Publication V: The Virtual Factory: Hologram-Enabled Control and Monitoring of Industrial IoT Devices.

I came up with the idea for the paper as a foundation for the integration of IoT devices with augmented reality (AR) interfaces. I contributed to the system design aspects and research foundation while Oleksii Moroz implemented and evaluated the system.

Publication VI: Edge Chaining Framework for Black Ice Road Fingerprinting.

I came up with the idea for the paper as a foundation for the applicability of our unikernel framework in the domain of black-ice detection. I have designed, implemented, and evaluated the entire system.

Publication VII: UIDS: Unikernel-based Intrusion Detection System for the Internet of Things.

I came up with the idea for the paper as a foundation for a unikernel - based intrusion detection system (IDS). I contributed to the system design aspects and research foundation while Nikolai Schwellnus implemented and evaluated the system.

Publication VIII: ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment.

I came up with the idea for the paper as a foundation for an orchestration framework enabling edge-cloud collaborative computing for road context assessment. I have designed, implemented, and evaluated the entire system.

1 Introduction

Many years of progress in the field of miniaturization of electronic components allowed to transform computers from large, bulky, room-sized machines to small devices fitting in our pocket. Similarly, computational power increased to the point where today's smartphones are many orders of magnitude more powerful than the first supercomputer, the IBM 7030 Stretch [9]. Today, the widespread proliferation of smart devices is fueling the so called digitalization process which is at the roots of the IoT era. Moreover, the line separating artificial from natural, as in humanity as biological lifeform, is blurred as Brain-Computer Interfaces (BCI) and sub-dermal microchips are becoming more mature [10].

The penetration of embedded devices in our daily life can be seen as a form of compute resources decentralization, where processing power is distributes across large networks of devices. In fact, recent studies suggest that there should be more than 50 billion devices connected to the Internet by 2025 [11, 12]. What we are witnessing can be seen as the *computing* equivalent of the everlasting struggle between centralization and decentralization forces, a recurrent theme in many aspects of human society (e.g., politics, energy generation) [13]. As an example, in 1980s a wave of decentralization led to a shift away from mainframes to PCs which culminated in decentralized systems using the peer-to-peer (P2P) technology.

Although super-computers are still very well in use, today we refer to them using the umbrella-term *cloud computing*. A stepping stone towards what can be seen as the Internet globalization, cloud computing revolutionized dramatically online services and applications offering an unprecedented model to offer and manage computational resources. As a matter of fact, cloud computing turned the odds in favor of centralized systems by concentrating enormous amounts of control, data and intelligence in remote datacenters. Today's cloud computing giants such as Google, Amazon, Microsoft etc., possess vast networks of large datacenters scattered around the globe serving millions of users. However, the recent proliferation of cyber-physical spaces pushed computational resources towards end-users. In contrast to cloud computing, this trend is in favor of a resource decentralization process, where hardware resources are not anymore remote and evanescent but, on the contrary, tangible and nearby. Figure 1.1(from [14]) shows the information age progress over multiple decades until the advent of cloud computing.

A meeting point between the resource centralization in the cloud and decentralization of the IoT is edge computing. The origins of edge computing lie in Content Delivery Networks (CDN) that were created in the late 1990s to serve web and video content from edge servers that were deployed close to users [15]. The modern concept of edge computing significantly expanded on this notion and, today, it is a network infrastructure

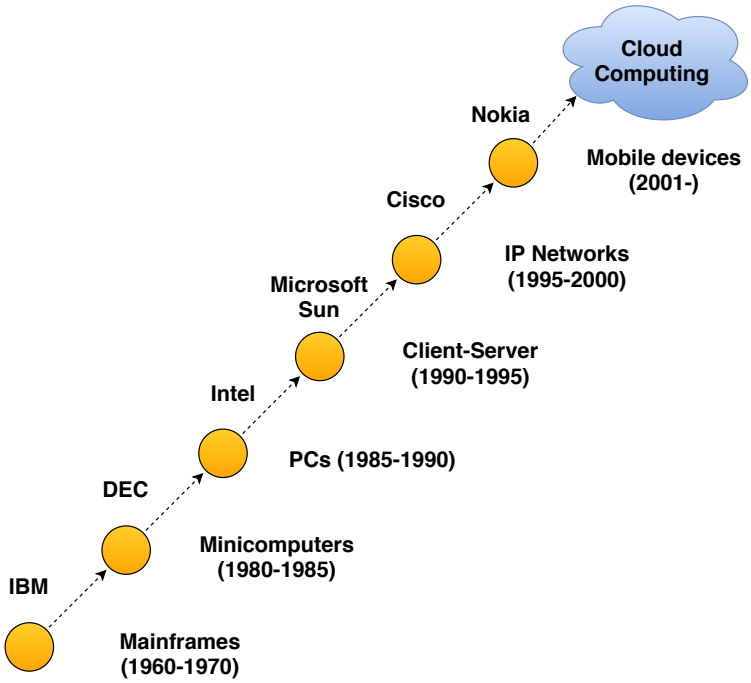


Figure 1.1: Information age evolution.

offering processing power and data storage to improve services in different ways, including reduced response time and bandwidth utilization.

However, there are different opinions regarding the actual placement of edge computing resources. Researchers have proposed different solutions in this regard including base stations, the mobile network, hotspots, and servers in the ISP infrastructure as a form of extension of a datacenter reach [16]. From the mobile network operator's perspective, edge computing should be integral part of the mobile network infrastructure and a key technology towards 5G adoption. Consequently, edge servers should be deployed at different locations, such as at the LTE base station site, at the 3G Radio Network Controller (RNC) site, at a multi-Radio Access Technology (RAT) cell aggregation site, and at the edge of the core network [17]. Alternatively, standard network routers potentially equipped with additional computational resources can be considered as part of an edge infrastructure and act as edge gateways [18, 19].

Research in edge computing has been driven primarily by the desire to either improve, or replace, the cloud computing model by pushing the application logic to the network periphery and closer to the datasource. The work presented in this thesis focuses on the importance of the interplay between edge and cloud, with, at its core, the trade-off between data transmission and computational latency. While the cloud has potentially unlimited computational resources, the additional latency introduced when uploading data from IoT and consumer devices can be especially detrimental for time-sensitive applications. Conversely, while offloading latency at the edge is negligible when compared

to the cloud, the computational delay and cost is higher. These observations call for a deeper understanding of the application logic semantics in order to repartition and split a complex software workflow into smaller sections that can be allocated to different network segments (e.g., edge, cloud). This has been explored both in industry and academia especially in the context of different virtualization and service decomposition techniques [20, 21]. Edge computing offers the possibility of leveraging the combined computational power of a large number of small devices by creating distributed execution chains. This is very similar to the idea of computing streams which are defined as a sequence of functions/computing applied on the data along the data propagation path [22]. However, this requires proper planning through resource management algorithms based on different metrics such as latency, energy cost, bandwidth, and hardware/software specified limitations to send opportunely the functions/computing [23, 24, 25, 26].

In this thesis, we aim at systematically addressing different edge computing challenges following a bottom-up approach. At the core of the research effort, there is the quest to find the meeting point between cloud and edge by taking the best of both worlds in order to improve existing applications and enable new services for end-users. Future edge computing deployments will succeed by co-existing with the traditional centralized cloud infrastructure if supported by proper cooperation and coordination mechanisms.

1.1 Problem Statement

The quest to correctly position edge computing as an emerging technology in relation to current trends is at the core of many research efforts in the field. Cloud computing has been thriving both in the research and industry domain by amortizing dramatically Capex costs in terms of infrastructure and software tools for businesses of any size. From the research perspective, it sparked interest in many directions including routing, resource management, virtualization, security, privacy. Currently, edge computing is receiving progressively more attention by the research community and industry players [27, 28, 29]. However, industry efforts are focused on using edge computing as an extension of the existing DC infrastructure and to be deployed inside ISPs networks in order to optimize the response time of cloud services. Regardless of the growing interest towards edge computing, it is still struggling to take off for many reasons including technical, business and deployment aspects to be overcome. Irrespective of the final decisive factors, we can identify a number of technical challenges including service programmability, service discovery, lack of standards, service orchestration, and privacy/security. In particular, the deployment and management of an edge infrastructure is, technically, more challenging when compared to the cloud, for the reasons described below.

- **Constrained Hardware.** Cloud datacenters are composed of sets of powerful servers with plenty of storage and computational resources paired with reliable network infrastructure. Additionally, the presence of many standards and best-practices guiding the datacenter development process facilitated greatly their success [30]. On the other hand, edge computing is often built upon heterogeneous, resource-constrained devices with may have poor connectivity. This makes not

only the deployment, but also the management of the computational resources harder as the involved devices are more prone to failures. In order to simplify the deployment of edge-cloud applications, it is important to efficiently partition and make use of the available hardware and network resources.

- **Scalability.** Distributed systems are at the foundations of modern wide-area services, offering different levels of abstractions to applications while hiding the challenges of running over a distributed set of heterogeneous devices. Under the hood, a distributed system manages multiple resources (e.g., computation, storage) across a range of devices, to hide latency, cope with malfunctions, and achieve scalability. The latter is harder to address in edge-cloud systems for multiple reasons: data fragmentation, heterogeneity, latency-sensitive applications and so forth. For example, data management and access is steadily growing in complexity as information becomes scattered across the network including: across devices in a single location (e.g., a data center), across multiple data centers, and at the points of collection (e.g., a self-driving car or a mobile phone) [31]. Additionally, scalability extends also to resource provisioning issues which become critical in multi-tenant environments. This calls for solutions where applications are automatically partitioned and distributed in order to maximize or improve the service quality.

- **Maintenance and Management.** Resource centralization in cloud computing helps in performing routine maintenance operations. The hardware is easier to access as it is not distributed across a wide geographical area. Edge computing is distributed in nature which noticeably complicates maintenance operations due to the physical distance between devices. In fact, in case of hardware faults, replacement requires direct intervention *in situ* which is comparatively more challenging and time-consuming than in a datacenter. Predictive maintenance, constant infrastructure monitoring, redundant deployment for critical units are example solutions to mitigate the maintenance burdens [32, 33].

- **Vulnerability.** Every cloud architecture has different privacy and security concerns. However, the cloud attack surface is smaller compared to the edge due to its centralized nature. Edge systems face additional security risks when it comes to data storage, authentication, and access control to mention a few. As a matter of fact, compromised network appliances, including edge devices, continue to be one of the most effective attack vectors for advanced threat actors. Unlike hosts that receive significant administrative security attention and for which security tools such as anti-malware exist, network devices are often working in the background with little oversight — until network connectivity is broken or diminished [34]. Securing the perimeter defense of edge computing devices is a problem that, if not addressed, can severely hinder also the datacenters these devices are connected to. In fact, once an adversary compromises a set of edge devices, it can gain full control of the network infrastructure eventually leading to traffic redirection, denial-of-service, data theft, or unauthorized changes to the data. Cloud services are as secure as the edge devices accessing them: if the latter are com-

promised, so will be eventually the former. Therefore, there is a need to develop security and monitoring tools which are compatible with and can be deployed on resource-constrained devices to strengthen the security of edge networks.

- **Virtualization & Application Model**. Paired with the widespread adoption of the many services offered by cloud providers, virtualization and containerization thrive as the resource management backbone of datacenter resources. This trend noticeably affected the development process of many applications fueling the progressive migration from monolithic software to micro-services architectures and service decomposition. While cloud datacenters can support any virtualization technology and, consequently, adapt to any application needs, the same is not necessarily true at the edge where hardware heterogeneity makes harder to find a *one-size-fits-all* solution. In connection with the points mentioned above, virtualization technologies play a key role in managing resource at the edge, abstracting hardware heterogeneity, limiting attack surface and fine-tuning the software deployment.

The aforementioned challenges summarize some of the roadblocks for edge computing to emerge as a disruptive technology which have been identified also in other research efforts [13]. We argue that edge computing will play a key role in empowering and enriching the services offered by the cloud by extending the reach of the latter beyond datacenters. This will bring benefits for both **latency-sensitive** and **latency-insensitive** applications by bringing compute resources closer to the data generators. Several early results from the research community in the domains of face recognition applications [35], wearable cognitive assistance [36], and application partitioning and offloading [25] demonstrated the potential benefits of the edge computing paradigm.

The problem we tackle is multi-faceted as the challenges we discuss are at the intersection between system design, resource provisioning at scale, and application requirements. Furthermore, the resulting solutions need to be compatible with traditional cloud computing technologies such that existing cloud-based applications can be ported and integrated to edge environment with minimal overhead. Moreover, this would facilitate knowledge transfer for developers which would be able to naturally make use of the edge-cloud infrastructure.

1.2 Methodology

The research problems investigated in this thesis are centered on the challenges of integrating edge and cloud computing and finding means to facilitate the success and acceptance of the former. Specifically, we aim at exploring novel virtualization techniques able to support service decomposition and fine-grained computation offloading for resource-constrained devices. The latter promotes offloading of small virtual instances which have reduced footprint and execution boundaries granted by their minimalist code surface. Resource limitations at the edge in terms of computational power and storage call for strict resource optimization procedures especially when multiple services are competing

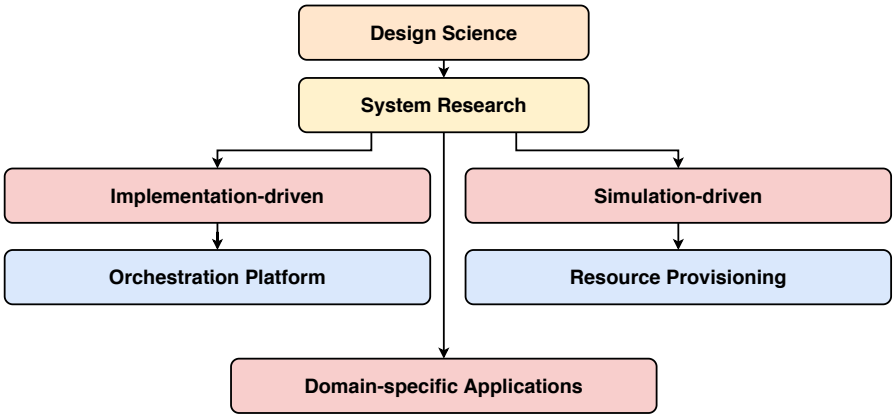


Figure 1.2: Methodology outline.

for them. Additionally, we look at problem at scale of edge-cloud infrastructures in the context of resource provisioning especially for real-time applications. In both cases, an extensive exploration of related work is conducted in order to identify gaps in past research to be addressed.

The twofold nature of the research conducted in this thesis required both an implementation and simulation-based approach to the problem, as shown in Figure 1.2. The former is necessary in the process of designing and developing from the ground-up a novel orchestration framework for unikernels. This is in line with the principles of Design Science (DS) research described in [37]. Specifically, we follow the steps advocated in the study: problem identification and motivation, objectives for a solution, design and development, evaluation, and communication. Hence, we first conceptualized and implemented the required software components before performing a quantitative analysis necessary to evaluate the effectiveness of our approach. As we frame our research as *systems research*, we consider acceptable the liberties taken in the process of structuring our framework which might naturally diverge similar approaches in past literature. However, we compared our solution against existing tools in order to validate our research and results. Moreover, all the software produced and used in the publications that are part of this thesis is openly available. Hence, experiments can be reproduced by other researchers in the spirit of supporting reproducible research in computer science [38].

For the resource provisioning research, we shifted our methodology towards a simulation-based approach. This is common practice when testing algorithms at scale due to the massive number of device required for running the experiments which would be infeasible to deploy in a real-scenario. Nonetheless, the data used in the simulations were generated from experiments on physical devices using tools provided by the manufacturer or extracted from past research measurements. As above, the simulator code and all the data are open-source so that the results can be replicated by other researchers.

1.3 Where Edge and Cloud Meet

Having framed the research direction and methodology in the previous sections, we focus on three facets of our approach: orchestration platform, resource provisioning, and domain-specific applications.

- **Orchestration Platform.** As mentioned before, cloud computing success was also fueled by the widespread adoption of virtualization technologies used to encapsulate services and flexibly scale those as a function of the utilization demand. Virtualization technologies are crucial for the success of edge computing as they can help in addressing the intrinsic heterogeneity found at the edge due to the presence of different types of hardware platforms. A virtualized platform can also facilitate the interoperability between cloud and edge by enabling transparent migration of applications across the infrastructure. Additionally, virtualization enables resource provisioning, isolation and partitioning among concurrent services. The research questions prevalent in this layer are as follows.

  RQ 1. What virtualization technique is feasible to be employed to match the requirements of both edge and cloud?
  RQ 2. What model of computation should be adopted to support the interplay between edge and cloud in order to support different classes of applications?
  RQ 3. How to support stateful services in fault-prone and resource-constrained edge networks?

  RQ1 focuses on understanding how existing and emerging virtualization technologies can be adapted or leveraged at the edge. Specifically, we focus on Lightweight Virtualization (LV) and its trade-offs. RQ2 build on top of the previous question and aims at formalizing a model of computation where edge and cloud collaborate to perform tasks submitted to the compute infrastructure. Finally, RQ3 expresses the need to empower LV to support stateful applications so that intermediate results are not lost in case of malfunction or resource starvation. Together, RQ1 to RQ3 aim at identifying the building blocks of an edge-cloud orchestration platform based on LV and a model to partition, distribute, and chain applications scattered between edge and cloud.

- **Resource Provisioning.** In order to serve massive numbers of users and applications, proper management of hardware and software resources is necessary both in the cloud and at the edge. Depending on requirements such as reliability, availability, performance, and scalability of the application, users are allocated to different servers to maximize their Quality of Experience (QoE). As resources are centralized in the cloud, their monitoring and controlling is less challenging than at the edge. To fully enable the interplay between edge and cloud, it is necessary to extend resource provisioning mechanisms to take into account both datacenters and edge devices alike. Compared to the orchestration challenges discussed above, we here abstracted from the implementation details and take a more theoretical

standpoint. The discussed challenges are summarized by the following research question.

RQ 4. How to allocate resource efficiently in an edge-cloud infrastructure to improve QoE for end-users?

RQ4 focuses on analyzing the characteristics of a multi-tier, edge-cloud infrastructure in order to properly allocate users to different classes of devices. In particular, we focus on performance degradation in multi-user environments where multiple applications compete for resources. Additionally, we look at the trade-offs of solving this problem from different perspectives.

- **Domain-specific Applications.** One of the major challenges towards adopting edge computing and its integration with the cloud paradigm is finding its *killer-app*. In other words, identifying the application domain(s) where edge computing can bring definitive benefits still remains an unanswered, pivotal question for the current research in the field. In our approach, we classify applications as latency- and non-latency-sensitive. Under the former, we group typical IoT applications of which some examples are sensors fusion, data aggregation, command-and-control systems. For the latter, we looked into multimedia-based applications such as augmented reality.

Figure 1.3 summarizes how the research questions mentioned above are covered in the publications attached to this thesis. The goal is to have a progressive introduction of the research presented in this thesis by moving from exploratory work in specific applications domains towards system design/architecture challenges in order to identify the current and future potential of edge computing.

1.4 Contributions

This thesis identifies critical integration challenges between edge and cloud computing to create an homogeneous infrastructure abstraction in spite of the apparent duality between the two technologies. We explore both fundamental challenges and domain-specific problems by providing technical solutions for each one of them. The main contributions are as follows:

- We introduce the concept of Lightweight Virtualization (LV) technologies which we see as an enabler in the process of integrating the edge and cloud infrastructures. Specifically, we discuss and compare the applicability of two different LV technologies as platforms for enabling scalability, security and manageability required by emerging edge-cloud applications. Additionally, we present open problems and highlight future directions to serve as a roadmap for both industry and academia.
- We devise an edge-cloud architecture supporting offloading of compact, single purpose tasks at the edge of the network for a variety of IoT and cloud services. The design principles behind the system we developed (FADES) are meant to

efficiently exploit the resources of constrained edge devices through fine-grained computation offloading. Subsequently, we expand our system with a computational model supporting the execution of distributed tasks called *pipelines* which require the cooperation of multiple edge devices.

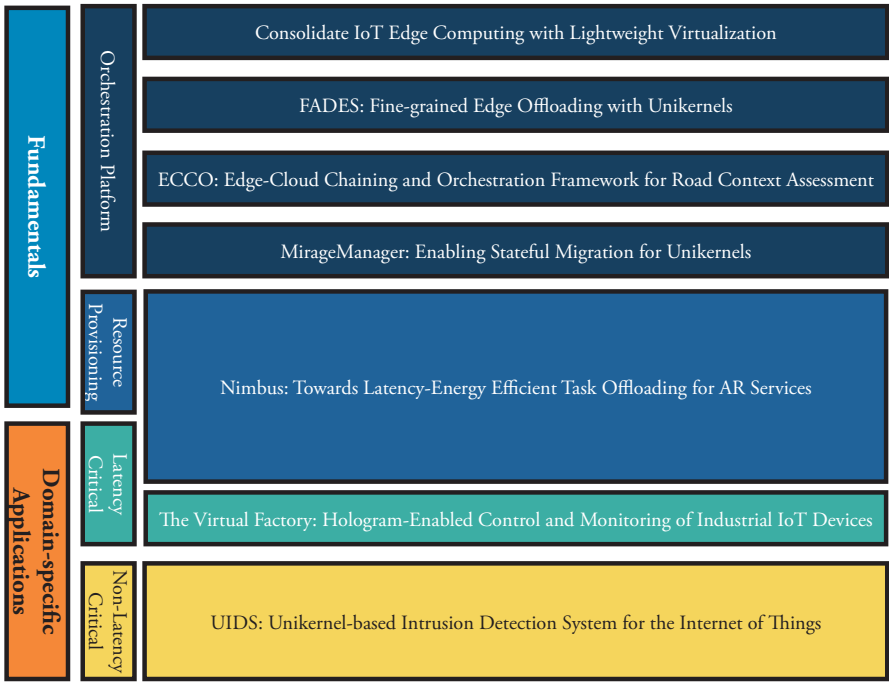


Figure 1.3: List of publications grouped as fundamentals and domain-specific paired with related research questions (RQ).

- We approach the problem of service migration in an edge-cloud infrastructure for stateful applications. This is a crucial problem in a distributed system that wants to be fault-tolerant or provide a form of dynamic load balancing. Most service decomposition technologies like lambda functions or containers either only support stateless applications or offer rudimentary migration tools. Similarly, LV technologies do not support stateful migration at all. To bridge this gap, we develop a system developed from the ground-up to perform migration while preserving the internal application state.

- As part of the resolution of domain-specific challenges, we look at two different classes of applications. In the domain of non latency-critical applications, we developed a lightweight Intrusion Detection System (IDS) for constrained devices. For multimedia applications, we designed and implemented an cyber-physical Augmented Reality (AR) application combining smart devices with holographic interfaces for industrial use-cases.

- Finally, we look at task scheduling and allocation challenges which are common in multi-tenant, distributed system like an edge-cloud infrastructure. Specifically, we analyze how to improve the QoE of mobile latency-critical applications by offloading computationally intensive tasks to the edge-cloud infrastructure. In particular, we focus on high-concurrency scenarios where many users compete for a limited set of resources offered by the infrastructure. We provide a latency-energy efficient allocation algorithm to provision resources for tasks offloaded by mobile devices. Additionally, we provide a fully-customizable and open-source simulator.

The research reported in this thesis encompasses the work published in eight original, peer-reviewed articles which also include additional contributions that are not covered in details in this thesis:

[**1**]: Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jörg Ott. "Consolidate IoT Edge Computing with Lightweight Virtualization." *IEEE Network 32*, no. 1, pp. 102-111. 2018.

[**2**]: Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. "FADES: Fine-grained Edge Offloading with Unikernels." *In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, pp. 36-41. 2017.

[**3**]: Vittorio Cozzolino, Oliver Flum, Aaron Yi Ding, and Jörg Ott. "MirageManager: Enabling Stateful Migration for Unikernels." *In Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*, pp. 13-19. 2020.

[**4**]: Vittorio Cozzolino, Tonetto Leonardo, Nitinder Mohan, Aaron Yi Ding, and Jörg Ott. "Nimbus: Towards Latency-Energy Efficient Task Offloading for AR Services" *IEEE Transactions on Cloud Computing*, 2022.

[**5**]: Vittorio Cozzolino, Oleksii Moroz, and Aaron Yi Ding. "The Virtual Factory: Hologram-Enabled Control and Monitoring of Industrial IoT Devices." *In 2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pp. 120-123. 2018.

[**6**]: Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. "Edge Chaining Framework for Black Ice Road Fingerprinting." *In Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, pp. 42-47. 2019.

[**7**]: Vittorio Cozzolino, Nikolai Schwellnus, Jörg Ott, and Aaron Yi Ding. "UIDS: Unikernel-based Intrusion Detection System for the Internet of Things." *In Workshop on Decentralized IoT Systems and Security (DISS)*, 2020.

[**8**]: Vittorio Cozzolino, Aaron Yi Ding, Richard Mortier, and Jörg Ott. "ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment." *In*

2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), pp. 223-230. 2020.

1.5 Thesis Structure

The thesis is organized as follows. Chapter 2 provides the necessary background knowledge to the thesis. Specifically, it discusses the state of the art in both cloud and edge infrastructure models followed by a set of scenarios where the interplay between the two can be leveraged to improve existing services. Then, specific challenges related to the integration between cloud and edge are introduced. Chapter 3 presents the domain-specific applications. We discuss non-latency critical scenarios dominated by IoT use-cases especially in the domain of security. Further, we look at latency-critical applications such as AR in industry settings. Chapter 4 introduces the main contributions on which the research presented in this thesis is based. In particular, lightweight virtualization technologies and task allocation algorithms with a focus on their applicability to edge computing environments. Finally, Chapter 5 concludes the thesis with a summary of solutions and outlooks for future work.

2 Background

This chapter provides the necessary background on edge computing by starting with a brief introduction of standard cloud infrastructures. After introducing the main components and design of a traditional datacenter, we describe in detail how edge computing came to be and what are the key ideas behind it. Further, we identify a set of application domains where edge computing can be a key-enabler technology. The chapter includes a discussion on several edge-cloud architectures proposed in past research and the design choices which make them suitable for specific application operation.

2.1 Cloud Computing Model

The core aspects of cloud computing have been described by the definition provided by the National Institute of Standard and Technologies (NIST) [39]: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Cloud computing has hugely impacted over the IT industry by providing virtually unlimited storage [40] and processing capabilities at low-cost enabling the realization of a new computing model where virtualized resources can be leased in an on-demand fashion, being provided as general utilities [41]. Almost all tech giants (like Google, Amazon, Dell, Apple, Facebook, etc.) widely adopted this paradigm in order to deliver services over the Internet, gaining increasing technical and monetary benefits.

The cloud can be split into four layers: datacenter (hardware), infrastructure, platform, and application. *Datacenters* (DC) are large-scale, distributed network systems built upon on a set of interconnected servers. Managing a DC is a non-trivial task and implies practical challenges as airflow management, facility thermal control, and power distribution. On top of the hardware part, cloud providers classify their services based on a layer concept. In the upper layers of this paradigm, Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) are stacked [42]. SaaS refers to the provisioning of applications running on Cloud environments. Resource rental can be done at different granularity, e.g., per server, and following various policies that provide different degree of freedom and responsibility. For example, PaaS refers and targets to platform-layer resources (e.g., operating system support, software development frameworks, etc.). IaaS refers to providing processing, storage, and network resources, allowing the consumer to control the operating system, storage and applications.

Another classification can be applied to cloud providers as described in [43]: (i) Private Cloud provisioned for exclusive use by a single organization, typically owned, managed,

and operated by the organization itself; (*ii*) Community Cloud — provisioned for exclusive use by a specific community of consumers that have shared concerns; (*iii*) Public Cloud — provisioned for open use by the general public; (*iv*) Hybrid Cloud — composition of two or more distinct Cloud infrastructures (private, community, or public); (*v*) Virtual Private Cloud—alternative aimed at addressing issues related to public and private clouds, taking advantage of Virtual Private Network (VPN) technologies for allowing business owners to setup required network settings (e.g. security, topology, etc.).

Regardless of the widespread adoption of cloud technologies, several technical and business-related issues are still unsolved. Specific issues have been identified for each service model, which are mainly related to security (e.g., data security and integrity, network security), privacy (e.g., data confidentiality), and service-level agreements, which could scare away part of potential users [44].

2.2 Edge Computing Model

While cloud computing has been dominating the Internet services delivery model since a few decades, the forces driving centralization are not the only ones at work. New services and applications for mobile computing and the IoT are pushing towards compute resource dispersion. This trend is motivated by multiple reasons including: (*i*) reduction of network latency for delay-sensitive application services (e.g., AR, VR, video analytics), (*ii*) data aggregation and sensor fusion required by the proliferation of IoT and smart devices in order to reduce uplink bandwidth utilization and optimize response time in critical situations, and (*iii*) better customization through context-awareness of location-dependent services. Edge computing is a paradigm in which substantial computing and storage resources are placed at the edge of the network, in close proximity to mobile devices or smart sensors.

The roots of edge computing reach back to the late 1990s, when Akamai introduced CDNs to accelerate web performance [45]. With time, CDNs expanded to deliver multimedia content, because the bandwidth savings from caching videos at the edge can be substantial. As edge computing gained more traction, the concept of CDN evolved again: instead of being limited to caching web content an delivering multimedia streams, it can also run arbitrary code just as in cloud computing. This code is typically encapsulated in a virtual machine (VM) or a lighter-weight container for isolation, safety, resource management, and metering [45]. Additional details regarding different virtualization technologies will be presented in Chapter 4.

2.2.1 What and Where is the Edge?

In past research, multiple solutions have been proposed to describe *what* is and edge network and *where* is physically located. We hereby discuss five main approaches: cloudlets, ad-hoc cloud, fog computing, MCC and MEC.

- **Cloudlet.** The work from Satyanarayanan et al. [46], proposed in 2009, pioneered the concept of bringing the computation/storage closer to the end-users through

cloudlets. The idea behind cloudlets is to place devices with high computation power at strategic locations in order to provide both computational resources and storage for the end-users in vicinity. Additionally, they are designed with the intention to support cloud virtualization technologies/protocols and deployable directly by a cloud provider. One of the main advantages of a cloudlet lies in its limited deployment footprint, allowing it to considerably reduce the user access latency compared to a remotely deployed cloud service.

- **Ad-hoc Cloud.** Another option to enable edge-cloud computing is to exploit the computational resources of the end-user devices through the so called *ad-hoc cloud*. The main idea is to combine the computational power of multiple end-user devices in proximity to process high demanding applications locally [47, 48, 49, 50, 51]. The key advantage of this approach lies in its proximity to users and sensors supporting the low-latency requirements of specific applications such as live video streaming, unmanned drone control. However, as most ad-hoc-based solutions, there are multiple challenges to be addressed: (*i*) discovery of nearby devices and network formation [52, 53], (*ii*) coordination between devices for task allocation [54], (*iii*) energy impact on each device [55, 56], and (*iv*) security/privacy issues [57, 58, 59].

- **Fog Computing.** A more general concept when discussing the edge computing model is known as *fog computing* paradigm (commonly called Fog). It was introduced in 2012 by Cisco to enable the processing of data generated by applications on billions of connected devices at the edge of network. Fog is a virtualized platform for managed compute resources that are colocated with devices deployed within the access network, e.g. routers, switches, access points etc. [60]. Fog computing can be seen as one key enabler of the IoT and big data applications [61, 62]. In fact, it offers: (*i*) low latency and location awareness, (*ii*) widespread geographical distribution compared to the cloud, (*iii*) large network of nodes, and (*iv*) support for real-time multimedia applications. Additionally, the characteristics of fog computing can be exploited also in other application domains such as healthcare, smart vehicles, and blockchain based communication protocols [63, 64, 65, 66].

- **Mobile Cloud Computing (MCC).** Known also as Cloud Radio Access Network (C-RAN), it is another alternative where cloud capabilities are embedded directly into the mobile network [67, 42]. The C-RAN leverages the idea of a distributed protocol stack, where some layers of the protocol are moved from distributed Radio Remote Heads (RRHs) to centralized baseband units (BBUs). The BBU's computation power is, then, pooled together into virtualized resources which are generally used for baseband processing but may also be used for the computation offloading to the edge of the network [68]. Similarly, MCC's primary objective is to provide a one-hop computation offloading facility for mobile subscribers, directly at the cell tower. MCC supports cloud virtualization technologies enabling a smooth integration with centralized solutions.

- **Mobile[1] Edge Computing (MEC).** The integration of edge computing into the mobile network architecture has been explored also by the Industry Specification Group (ISG) within European Telecommunications Standards Institute (ETSI)[17]. The solution described by ETSI is known as MEC and it is driven by a strong standardization efforts by prominent mobile operators (e.g., Vodafone, TELECOM Italia) and manufactures (e.g., Nokia, Ericsson, Huawei, Intel). The main goal of the ISG ETSI is to integrate cloud computing resources seamlessly into the mobile network by facilitating the operations of the different involved parties (mobile operators, service providers, vendors, and users). However, MEC also extends to non-mobile resources. In fact, it incorporates a wide, non-homogeneous variety of compute devices including desktop PCs, tablets and micro-datacenters.

  The concept of MEC is often the model considered to be representative of a standard edge computing architecture and it is the one we follow in the research presented in this thesis. In particular, we focus on the non-mobile part of the edge which is composed of resource-constrained devices equipped with different sensors. We extensively explore this *part* of the edge *ecosystem* as a place where to preprocess raw sensor data and deploy computation in the form of small, virtualized applications. More details about our contributions will be provided Chapter 4 and 3.

The majority of the edge computing architectures explored in research is based on a three-tiered infrastructure, as shown in Figure 2.1. As expected, the edge layer is closer to the end-devices compared to the cloud. Thus, even though the former has less computational power than the latter, it can deliver a better Quality of Service (QoS) by offering lower latency due to the physical proximity to the compute nodes [69]. In fact, the edge computing paradigm embeds compute nodes into the network, differently from the cloud perspective where they are concentrated in DCs. Cloud servers are deployed farther away from the end-users leading to significant increase in the network communication delay. On the other hand, cloud servers have potentially more computing power and data storage at their disposal which can be used to provide massive parallel data processing. Examples of applications that can benefit from it are big data mining, big data management, and machine learning [70, 71].

The edge layer encompasses heterogeneous devices offering different resources such as real-time data processing and data caching. They are deployed in proximity of the end-devices and handle most of the traffic flowing at the edge of the network. By leveraging this intermediate layer, end-users can benefit of much better performance on data computing and storage by paying a discounted price in terms of latency compared to accessing cloud DCs.

The end devices can be categorized in fixed and mobile. The former include smart sensors and actuators which are, for example, part of the smart infrastructure; the latter are handheld devices or mixed reality headsets. Although these devices are getting increasingly more powerful in terms of hardware capabilities, new mobile applications

[1]The adjective *mobile* is often replaced with other terms in related literature.

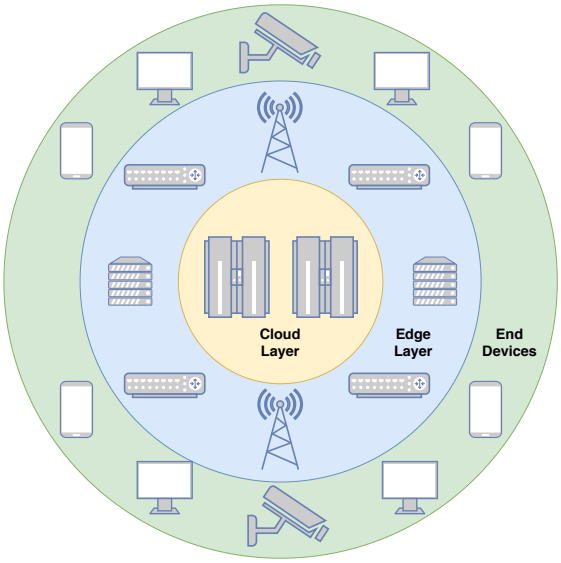


Figure 2.1: Edge-cloud layers structure.

are getting progressively more sophisticated requiring complex processing, in a short time window, and for a prolonged amount of time. Consequently, this is also a problem when considering the energy requirements of interactive multimedia applications such as AR. In fact, these are power-hungry and can quickly drain the phone's battery [72]. Fixed devices in the class of sensors or small embedded boards that are part of the smart infrastructure are often resource-constrained and require support from more powerful machines to perform their computation. For example, *feedback control systems* [73] are widely used for *automatic control* [74] loops of many electromechanical systems found, for instance, in industrial processes. They are based on the principle of measuring quantities through sensors which lead to changes to the system by means of actuators. Depending on the complexity of system, additional processing power might be required to perform advanced sensor fusion or data aggregation/manipulation steps. Therefore, for both these categories, computation offloading at the edge or in the cloud, depending on the task complexity and requirements, is needed to support a variety of applications.

In the next Section, we will discuss application domains where and cloud and edge computing can be used in tandem to enhance current services and potentially enable new ones.

2.3 Edge-Cloud Applications

Over the past few years, several parts of the computer science research community have started to explore effective ways to analyze data spread over multiple locations. In some cases, datasets are collected from multiple locations, such as sensors (e.g., mobile phones and street cameras) spread throughout a geographic region. Then, these data need to be processed close to where it is produced in order to meet different application de-

mands such as: high reliability for critical infrastructure control and self-driving cars, low latency in real-time multimedia applications (e.g., AR, video analytics), low cost for sensors data fusion, and data privacy in crowd computing and Multiparty Computation (MPC) [75]. Additionally, some distributed applications that do have specific characteristics, such as having distributed, non-overlapping, large datasets spread across multiple nodes in the network, can naturally benefit from the edge-cloud paradigm by distributing and spreading the compute load similarly to other approaches used in large clusters (e.g., map-reduce [76]). Often, some portion of the analysis may take place on the end-host or edge-cloud (to respect user privacy and reduce the volume of data) while relying on remote clouds to complete the analysis (to leverage greater computation and storage resources) [77].

Although cloud computing powers the majority of today's services, the aforementioned applications would benefit by shifting from the traditional cloud model to an hybrid edge-cloud one. We select a subset of scenarios that can benefit from the edge computing model.

- **Deep Learning Applications.** The increased processing power afforded by graphical processing units (GPUs), the enormous amount of available data, and the development of more advanced algorithms has led to the rise of deep learning. Much of this growth is being driven by tech giants, such as Facebook, Apple, Netflix, Microsoft, Google, and Baidu [78]. Deep learning currently fuels manifold applications including computer vision [79, 80, 81], natural language processing [82, 83, 84], pharmaceutical research [85] and big data analytics [86, 87] among many others. Deep learning is based on computationally intensive processes required to train a model and then use it to perform inference. The latter requires considerable amount of compute power due to the potentially high dimensionality of the input data which exacerbates the problem of meeting the stringent real-time requirements required of modern multimedia applications. Training a deep learning model is also a computationally expensive task due to millions of parameters that need to be refined over multiple training steps.

  In order to meet the computational requirements of deep learning applications, a typical approach is to leverage cloud computing. However, there are multiple issues such as response time (especially for real-time applications), privacy, and scalability [88]. While the training time of neural networks remains an operation that does not affect the end-user experience, inference time strongly influences the responsiveness and smoothness of the application. Edge computing is a possible solution to address the aforementioned challenges. For example, the inherent proximity of edge computing's resources to data sources on the end devices amortizes end-to-end latency, enabling real-time services. Scalability challenges are addressed by the natively distributed and hierarchical structure of edge computing where end devices applications can be allocated to different nodes in the networks based on their requirements. This avoids transferring large amounts of data to remote DCs and, instead, leverages computational resources in proximity. Additionally, by pairing edge nodes with cloud DCs, compute resources can scale smoothly with

the number of clients, avoiding network bottlenecks at a central location. This is especially beneficial for deep reinforcement learning [89] and federated learning [90] that leverage the collaboration among edge nodes to exchange the learning parameters for a better training and inference of the models, and thus enabling dynamic system-level optimization and application-level enhancement [91]. From a privacy perspective, edge computing can help by supporting local data processing at trusted edge servers or devices avoiding transfer of potentially sensitive information across the public Internet [92, 93].

- **Smart Infrastructure and Automated Vehicles.** The combination of smart infrastructure and automated vehicles will change our commute and driving experience. The former provides a physical backbone combined with an information and communication technology (ICT) and compute network to collect raw data from the road and enhance existing services and systems such as: rapid transit, waste management, road and railway networks, traffic lights and so on [94]. The latter is steadily gaining more traction from both academia and industry with the promise of offering a safer, more convenient, and more efficient transportation system. Apart from onboard sensing, automated vehicles access many cloud services (e.g., high definition maps, dynamic path planning) through a mobile connection to precisely understand the real-time driving environments.

  However, these automated driving services, which have large content volume, are time-varying, location-dependent, and delay-constrained [95]. As safety is one of the primary concerns for automated vehicles, the ultimate challenge is to design an edge computing infrastructure to deliver enough computing power, redundancy, and security to guarantee it [96]. As an example, reducing the overall network latency to distribute critical information about the road conditions or hazards to vehicles could, in same cases, save lives or reduce damage to property. This can be achieved by empowering the smart infrastructure in a way that it can autonomously process raw data from road sensors and distribute information to vehicles.

- **Real-time Multimedia Applications.** Since the appearance of consumer mobile devices equipped with many sensors and powerful chipsets, multimedia applications have received increasing interest among smartphone users. Recent studies report that the mobile AR (MAR) [97, 98, 99] adoption currently stands at 32%, where 54% of the respondents use mobile AR at least once per week and 36% percent several times per week [100]. However, MAR applications often rely on computationally intensive computer vision algorithms with extreme latency requirements. In fact, real-time applications require a Round Trip Time (RTT) ranging from 150ms to 500ms for online gaming and telemetry, respectively. In practice, a smooth AR experience requires a much lower latency in the range of 20ms to avoid phenomena such as motion sickness [101]. To compensate for the insufficient mobile computing power or to save on battery utilization, offloading to a remote device is often desired [102, 103]. Several research approaches focused on offloading intensive computer vision operations to cloud datacenters [104, 105, 106].

However, this introduces additional network delays which makes challenging to meet the low latency requirements of real-time multimedia applications. Edge computing can be a valid alternative to support AR applications because, on the one hand, RTT latency is lower compared to accessing cloud datacenters, on the other, edge servers can be equipped with powerful GPUs able to quickly perform complex computer vision tasks.

- **Industrial Automation.** Research advances in the past years allowed the introduction of Internet of Things (IoT) concepts in many industrial application scenarios, fueling the advent of the so-called Industry 4.0 or Industrial Internet of Things (IIoT) [107]. Industrial applications in the IoT domain require location awareness and low latency [108]. Due to limitations of the cloud platforms, different approaches considered the edge computing model as a valid alternative [13, 109, 110, 111] due to its intrinsic properties of locality and closeness to the datasource which helps in mitigating latency issues. Additionally, edge computing helps in the process of re-configuring smart factories, one of the needs of digital manufacturing enterprises, which must offer highly customizable products based on the customers requirements or adapt quickly to changes in the production process.

2.4 Challenges

The application scenarios mentioned above highlight how edge computing can play a crucial role in addressing some of the cloud limitations. However, the integration of two technologies presents multiple challenges including security, resource management, workload orchestration, deployment, and maintenance, to mention a few. Due to the new and dynamic nature of the researched topic, the proposed solutions for such problems might differ greatly from each other. This adds yet another facet to the problem: lack of clear standards to guide the development of a framework that addresses all the aforementioned problems. In this section, we focus on two interconnected aspects: orchestration and resource provisioning for which we introduce multiple state-of-the-art edge computing solutions with their limitations. The discussed solutions lie in the scope of the contributions presented later in Chapter 3 and 4.

2.4.1 Orchestration Platform

Orchestration is the automated configuration, management, and coordination of computer systems, applications, and services [112, 113]. Often, it is discussed in the context of virtualization and service provisioning for cloud DCs where it assumes the connotation of a workflow defining precise action towards larger goals and objectives. In the process of integrating edge and cloud, the concept of orchestration has to be revisited in order to support the heterogeneity of the infrastructure and make the best out of the available hardware resources. In this context, virtualization assumes a fundamental role as it abstracts the available physical resources and presents an abstract computing framework on which tasks can be easily shifted across the infrastructure. In fact, several researchers

assume that edge resources are virtualized and support different virtualization techniques ranging from traditional virtual machines (VM) [46], to containers [114, 115], to lambda-based serverless architectures [116, 117, 118]. Recently, a novel, lightweight, virtualization technique called *unikernels* has received increasing attention from the research community. Inspired by past works on library OSes [119, 120, 121], unikernels are single-purpose appliances that are compile-time specialised into standalone kernels. They were designed as a new approach to deploying cloud services via applications written in high-level source code [122]. We are unaware of any existing work exploring the applicability, trade-off, and orchestration challenges of unikernels in an edge-cloud infrastructure. Differently from traditional VMs orchestration which has been available and improved over many decades, with unikernels we are just at the beginning. Basic functionalities to manage, monitor and interact with unikernels are lackluster. Additionally, scaling assumes a different meaning with unikernels due to their impossibility to scale *vertically* (e.g., allocate more CPU cores) but only *horizontally* (e.g., more workers in parallel). More details will be provided in Chapter 4.

2.4.2 Resource Provisioning

Resource management is a crucial and necessary technique adopted in any system with limited available resources. Cloud DCs are often assumed to have *unlimited* resources but, in reality, this is the result of optimal and precise management of the available hardware and software resources. This is primarily achieved through tools based on predictive models or algorithms to profile running applications resource usage. Profiling is performed by analyzing a set of characteristics such as background workload, historic data, overhead etc. as a function of the specific profiling goal (cost, application or resource management) [123]. Additionally, these solutions must be able to scale with the massive amount of users accessing the DC and using potentially hundreds of thousands of VMs or similar virtualization technologies [124]. In this context, task assignment as a function of the resources in the cloud is a difficult problem to be solved, which requires efficient task scheduling algorithms [125]. In the process of integrating edge and cloud, necessary steps must be undertaken to incorporate edge devices into previously explored resource provisioning problems.

2.5 Summary

In this chapter, we described both the cloud and edge computing models highlighting core characteristics, shortcomings and differences between the two. Especially, we discussed the different flavors of edge networks proposed in past work while clarifying which one we embrace and adopt in our research. Then, we focused on application scenarios that could benefit from the interplay between edge and cloud as we see clear potential in an hybrid infrastructure. Finally, we described a set of challenges in the path of integrating this two technologies. In particular, we emphasized platform orchestration and resource provisioning that are the two research domains into which this thesis falls. For the former, we will focus on virtualization technologies which cover a crucial role in the

creation of an homogeneous edge-cloud infrastructure. For the latter, we will look into provisioning algorithms which, as already mentioned in this chapter, are integral part of cloud DC management operations and requires additional investigation to be ported to the edge. Both these core topics will be discussed later in Chapter 4. In fact, in the following chapter we will make a short digression by looking at scenarios where and how the edge computing model can help improving existing application.

3 Exploring the Edge Computing Potential

Chapter 2 introduced the multitude of edge computing models which were proposed in past research to address specific challenges and requirements. Such models support multiple use-cases of which some were already mentioned in Chapter 1. Here, we present two use-cases demonstrating the applicability of different technologies in combination with the edge computing model to improve or support existing services. Figure 3.1 summarizes the thesis structure and in this chapter we will focus on contributions in the *Applications* domain. Specifically, *UIDS* and the *Virtual Factory* which are exploratory works aimed at bringing to light the potential of edge computing in two different domains: network security and multimedia applications.

3.1 UIDS: Unikernel-based Intrusion Detection System for the Internet of Things

As mentioned in Chapter 1, one important aspect to take into account in the edge computing and IoT domains is security, which is the greatest risks of the uncontrolled proliferation of resource-constrained devices. Most manufacturers' top priority appears to be getting their product into the market quickly, rather than taking the necessary steps to build security from the start, due to high competitiveness of the field [126]. Additionally, IoT devices are often not powerful enough to run traditional security tools, which renders edge computing networks more exposed to attacks.

What is required are security tools that are lightweight, modular, and easily deployable. Hence, we bring forward the concept of *composable security* through LV, with the latter embedding self-contained security functionality that can be quickly deployed on-demand. In this section, we present UIDS: a signature-based intrusion detection system for IoT. Our prototype is based on the IncludeOS unikernel, ensuring low resource utilization, high modularity, and a minimalist code surface.

3.1.1 System Design and Implementation

An IDS is a system that monitors network traffic for suspicious activity and alerts when such activity is discovered. IDSes are classified primarily into signature-based and anomaly-based. The former uses knowledge of previous attacks to detect them. Hence, one downside is that new attacks cannot be detected as long as their signatures is not known in advance. As a result, frequent updates of the signatures database are required to keep the IDS effective. Anomaly-based IDSes require learning phase during which a *normal operation* model is constructed based on regular network traffic traces.

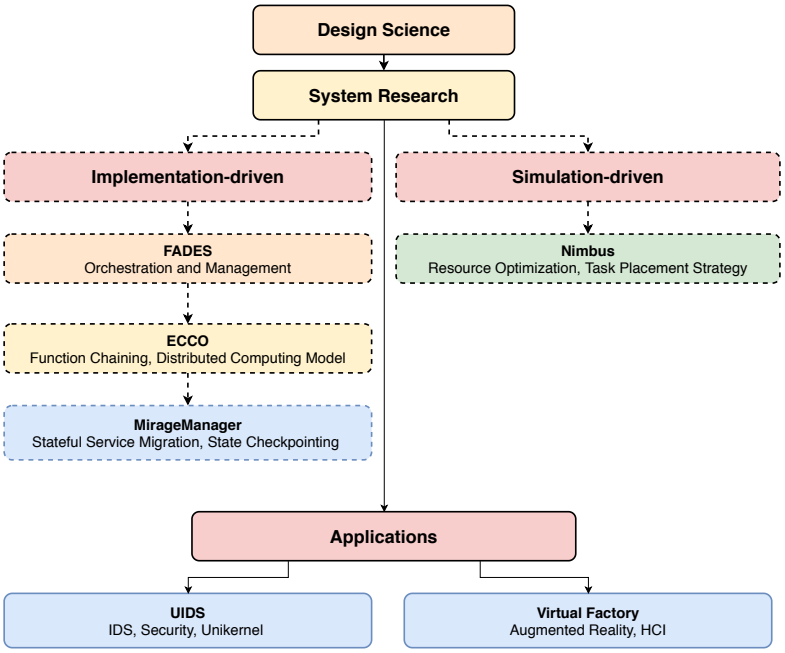


Figure 3.1: Domain-specific contributions.

The actual network traffic characteristics are then compared to this baseline, and if an anomaly is found, the IDS will generate an alert. Machine learning is often used to build models trained on non-malicious traffic. Incoming packets not fitting the model are classified as abnormal and an alert is generated. A downside of this approach is that no malicious traffic must be present during the model learning phase, otherwise the baseline would be compromised.

UIDS was designed as a signature-based IDS capable of detecting common DoS attacks, such as TCP SYN flood, TCP ACK flood, and UDP flood. Additionally, it can detect the most common port scans in three different variations: *one-to-one*, *distributed* and *decoy* scans. We implemented our prototype on top of the IncludeOS [127] unikernel which follows the *zero-overhead* principle and is written in C++.

UIDS builds on top of and expands the rudimentary connection tracking capabilities of IncludeOS in order to classify traffic as suspicious or benign. Additionally, it keeps state information regarding possible malicious packets. We take advantage of many useful features offered by the state-keeping functionality of IncludeOS for network connections, UDP and ICMP, and a more sophisticated one for TCP. In addition, its modular network stack allows us to easily capture packets on the wire and redirect them to custom modules for additional processing. Figure 3.2 shows the modifications made to capture packets and detect attacks.

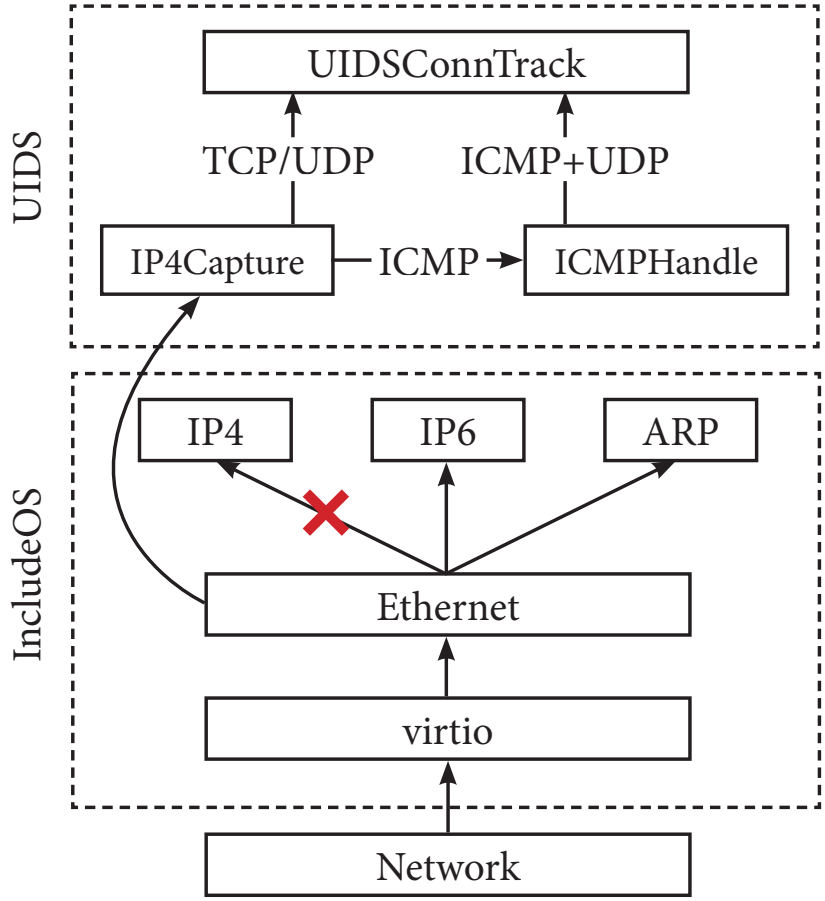


Figure 3.2: UIDS structure and packet reception path.

In order to detect malicious traffic, UIDS performs the following steps. First, packets received by the virtio device are passed up in the network stack hierarchy of IncludeOS. After the Ethernet layer, we redirect packets to a custom capture module bypassing the standard one as marked in Figure 3.2 with a red cross. Subsequently, we forward them to the core of our system: the *UIDSConnTrack* module which contains the complete logic to track suspicious packets. As this is a signature-based IDS, we use a set of rules to identify potentially malicious packets. The *UIDSConnTrack* module stores information about malicious packets on a per-host, per-port basis using *trackers*. The latter are implemented as unordered maps, saving the address of the sending host, in addition to the scan type and time. DoS detection is implemented similarly to port scan but with a different ruleset. In this case, trackers are simple packet counters, as common practice for such attacks, and store less information about the sender to save on resources. More details about the detection of malicious packets and data structures used in our system can be found in the research article attached to this thesis (Publication VII).

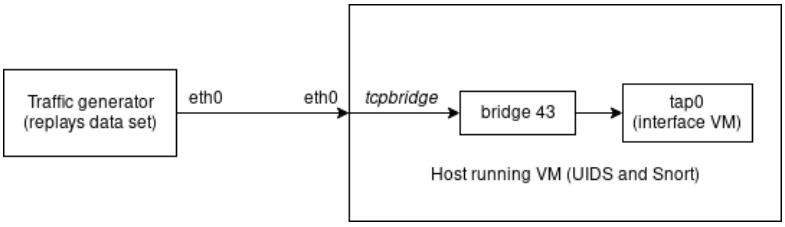


Figure 3.3: Setup for traffic replay.

3.1.2 Evaluation Setup

UIDS was benchmarked against different datasets normally used to evaluate the effectiveness of an IDS capabilities. These are often developed to train and test anomaly-based IDSs using machine-learning but can also be used to evaluate signature-based IDS. In our evaluation, we use two publicly available datasets containing DoS attacks and port scans in a packet-based format: TRAbID [128] and CICIDS 2017 [129]. In addition to existing datasets, we use a small-scale testbed of our design to stress-test UIDS. Both port scans and DoS flooding attacks are used to stress-test our implementation and evaluate the accuracy of alerts raised by UIDS and Snort [130]. Snort is also a lightweight intrusion detection system developed for small, lightly utilized networks. We compared our system against it because it is one of the most widely known network IDSes and has been used as reference benchmark in past literature [131, 132, 133].

Traffic replay. For this experiment, we run both UIDS and Snort on top of Kernel-based Virtual Machine (KVM) with Quick EMUlator (QEMU) using bridge networking to expose an interface to replay traffic to. We replay the dataset network traffic on the *traffic-generator* node and send it to the device hosting the IDSs via the incoming network interface, as shown in Figure 3.3. On the receiving host, traffic is forwarded to the bridge interface using the tool *tcpbridge* included in the *tcpreplay* tool suit. Finally, the bridge interface (*bridge43*) is connected each Virtual Machine (VM). Figure 3.3 shows the complete network architecture.

The CICIDS2017 traffic is split into port scan and DoS to reduce the evaluation period. However, such a procedure can introduce false-positives in the first parts of the network traffic since some connections might have been established right before the split. These false-positives were filtered out from the analysis. The TRAbID dataset provides two traffic captures for port scan and DoS. We use the *probe_known_attacks* capture for the evaluation of the port scan detection accuracy.

Live traffic. Besides testing UIDS against existing datasets, we also generated our own network traffic traces. To do so, we connect two hosts to a switch supporting port-mirroring. An additional host running UIDS and/or Snort is connected to the mirrored port, to intercepts all traffic generated between the hosts. Figure 3.4 illustrates the described setup. To generate the traces, we uses the tool *sourcesonoff*[1], which outputs realistic Internet-like traffic using statistical models, detailed in [134].

[1] `http://www.recherche.enac.fr/~avaret/sourcesonoff`

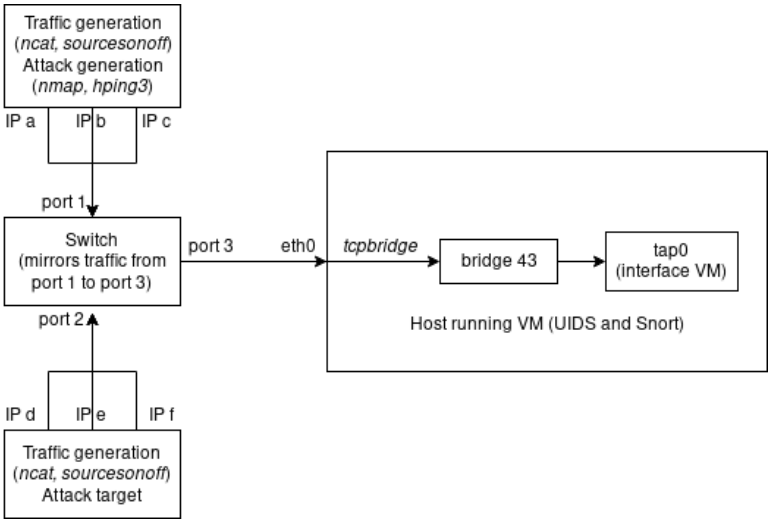


Figure 3.4: Setup for live traffic.

3.1.3 Results

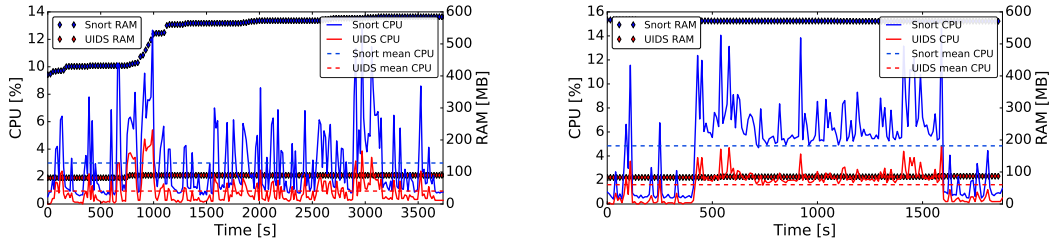
We now present some of the results obtained by testing UIDS with the datasets described above and using Snort as baseline. In particular, we report here only the results with the CICIDS2017 dataset while the ones for TRAbID and with the custom testbed can be found in the respective research article [7]. We focus primarily on CPU and RAM utilization to evaluate the compatibility of UIDS with resource-constrained devices.

The memory footprint of UIDS is only ≈2.3MB and it boots in some 200 ms on a non-optimized version of KVM (we did not use Solo5[2]). We use two hardware platforms in our tests: a laptop equipped with an Intel i7-4710@2.5GHz (LAP) and a Raspberry Pi 3B+ with an A53 ARMv8@1.4GHz (RPI). On the former, both IDSes run virtualized on top of KVM. On the latter, due to the lack of support for ARM, IncludeOS can not be directly virtualized using KVM. Instead, we emulate the x86 architecture on top of ARM using QEMU. As a result, UIDS suffers a performance penalty due to the additional emulation overhead which shows in the results as well. Conversely, Snort runs baremetal, which gave it a considerable advantage in terms of performance.

The CICIDS2017 results are divided into port scan and DoS attacks, and are described as follows.

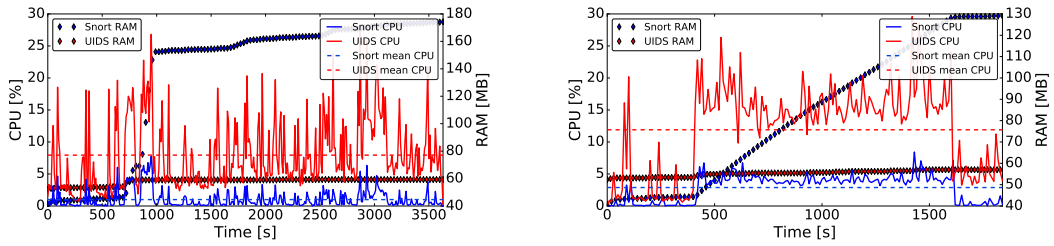
Port scan. Both UIDS and Snort are able to detect most TCP/UDP-based scans contained in the dataset. They are executed at specific time windows for which additional details can be found in the respective research article [7]. The CICIDS dataset supposedly contained FIN-, NULL-, and XMAS-scans, but we could not find any evidence of such scans in the downloaded dataset. The only difference in port scan detection between Snort and UIDS is the ICMP ping scan, which is not implemented, and therefore, not detected by UIDS. The TCP version and window scans have similar characteristics as

[2] `https://github.com/Solo5/solo5`



(a) **(b)**

Figure 3.5: UIDS vs. Snort (CICIDS2017, LAP) — Port scans (a) and DoS (b).



(a) **(b)**

Figure 3.6: UIDS vs. Snort (CICIDS2017, RPI) — Port scans (a) and DoS (b).

TCP SYN or connect scans, and are detected by both IDSs but classified as TCP SYN scans. Snort and UIDS generate false positive alerts for FIN scans in the first 120 seconds of the dataset. As mentioned previously, this is due to splitting the dataset which led both systems to see finalization packets that belonged to connections lost during the splitting.

Figure 3.5a shows the resource utilization of both systems. Overall, the CPU usage is low for both IDSs because the packet rates in the CICIDS dataset average around ≈330 *pps* and approximately 1 Mbps. Memory consumption is definitely higher for Snort, with 400-600MB, against UIDS, with less than 100MB (**4-6x** lower). A spike in memory usage can be seen after the first port scan is executed. This spike is modest for UIDS with a variation of ≤10MB but substantial for Snort with ≥130MB.

Flood-based DoS. The DoS attack contained in the CICIDS2017 datasets was generated with the open-source tool Low Orbit Ion Cannon (LOIT)[3]. Both Snort and UIDS emit alerts correctly during the active phase of the attack. Figure 3.5b clearly shows the beginning and end of the DoS attack in relation to the CPU utilization. As foreseen, memory consumption remain stable and marginal during the attack since the very little state information needed storing to detect flood-based DoS attacks. UIDS proves to be extremely lightweight compared to Snort in this benchmark. In fact, it allocates ≈**4x** less memory than Snort during the attack peak and on average **3x** less CPU.

[3] `https://sourceforge.net/projects/loic/`

Figures 3.6a and 3.6b show CPU and RAM usage for the Raspberry Pi for CICIDS2017 port scan and DoS traffic, respectively. While UIDS CPU usage is **5-6x** higher compared with Snort, memory allocation is reduced as we take advantage of the lightweight nature of unikernels. Hence, considering that we are running in an emulated x86 environment, UIDS can handle moderately fast traffic (up to ≈34Mbps) and reliably detect the same attacks also when running on an embedded board.

3.1.4 Discussion

UIDS is the first prototype of signature-based unikernel. In connection to what already discussed in previous chapters, our work shows the applicability of LV in the edge computing domain. UIDS showed great potential by delivering better resource efficiency, isolation, and a small memory footprint without sacrificing on the security aspects. In fact, it required **2-3x** less CPU and up to **8x** times less memory than Snort without influencing the detection capabilities.

The work presented in this section shows the potential of LV and suggests possible application domains where edge computing can be helpful. Instead of running complex, monolithic software stacks, with our approach we point in the direction *composable* security. Specifically, the use of unikernels allows to activate on-demand different security features on edge devices which can be stacked or composed as needed. Additionally, when combined with our unikernels framework (ECCO), the security functionalities deployed on different edge node can communicate and cooperate creating a distributed security tools network.

3.2 The Virtual Factory: Hologram-Enabled Control and Monitoring of Industrial IoT Devices

The foreseen massive diffusion of smart devices offers opportunities to build immersive human-computer interfaces where physical and virtual world blend. In domestic, industrial, and commercial settings, IoT offers innovative ways to interact with our surroundings. For example, at the core of Industry 4.0, there is the increasing digitalization of all manufacturing and manufacturing-supporting tools. This leads to an increasing amount of actor- and sensor-data supporting functions of control and analysis [135].

As machines and industrial physical processes change, the interfaces to interact with them should evolve and adapt. Today, most information about the state of physical processes is collected using Supervisory control and data acquisition (SCADA) systems and monitored by human operators. However, in the near future this might change dramatically due to the proliferation of VR/AR which can support human workers in a rapidly changing production environment. In fact, factory workers will be faced with a large variety of jobs ranging from specification and monitoring to verification of production strategies [136]. AR can help, for example, by providing a *virtual walk-through* to guide an inexperienced worker through unfamiliar tasks (e.g. assembly of new products) by visualizing information directly in the relevant spatial context [137].

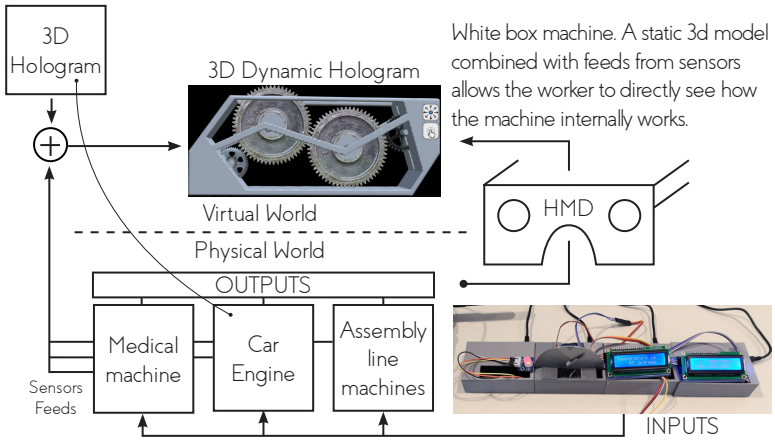


Figure 3.7: System overview.

AR headsets usually receive data wirelessly and in real-time or quasi-real time in order to provide an attractive user experience and avoid unpleasant symptoms such as motion sickness. In this scenario, edge computing can support in-situ operations by providing local computing power and low latency communication to observe the AR application QoS requirements.

In this section, we present our work on building a system and evaluate the extent to which AR can help to interact with complex machines through direct, visual, three dimensional (3D) feedback. Figure 3.7 shows that, with AR, the physical model of a machine is represented by inputs, outputs, and readings from sensors. The physical model becomes a virtual, dynamic model based on these parameters. Hence, a potential worker can actually see the way a machine works, given the availability of a 1:1 holographic model matching it.

3.2.1 System Design

Figure 3.8 and 3.9 show the three-layer architecture of the system and its workflow, respectively. The *IoT* layer is a network of IoT devices, such as smart sensors and actuators, which allow the user to interact with our system. The available physical devices have to register with our system and share their capabilities via a REST protocol. Hence, a bootstrapping phase is required to detect the available sensors and actuators in the network and associate them with a physical machine. To do so, a semantic representation of the device functionality is exchanged with the end-user layer and used for automated building of a holographic interface.

The *end-user* layer is the core of our system and provides the holographic abstraction of the physical world. The *edge* layer is primarily responsible for storage, administration, and organization of the network of sensors. It is built as an event-driven application and it is composed of four main modules. The *UI Manager* is responsible for automated generation of holographic interfaces based their semantic representation. The *Event Manager* stores the information about IoT devices and processes device detection and

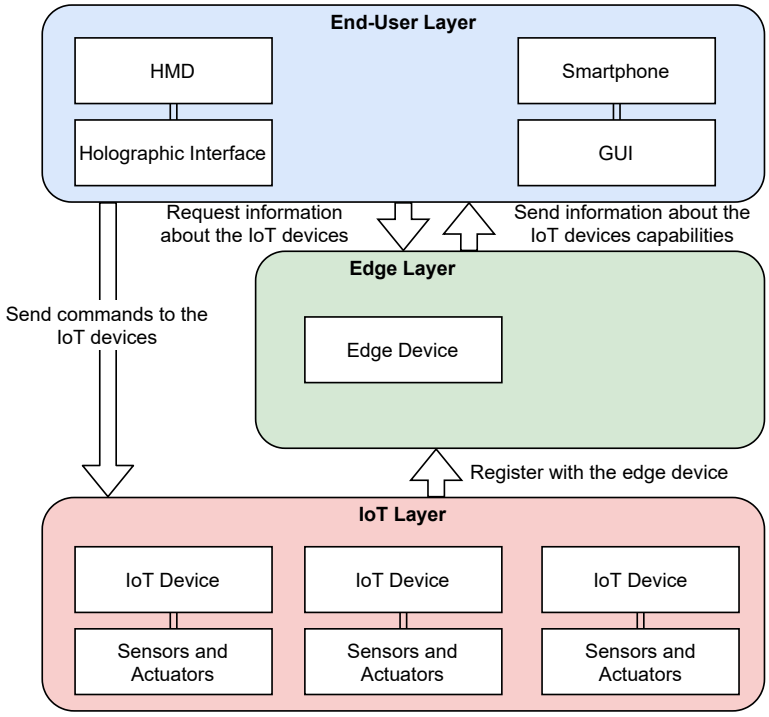


Figure 3.8: System design.

interaction events to update the UI Manager. The *Server Manager* is responsible for all data exchanges between the devices in the system. The *Semantic Module* is the data layer of the application and stores information about the IoT devices and their virtual twin plus the exposed functionalities.

3.2.2 Evaluation

We conduct two types of evaluations: a user study and application benchmarking. We prepare a room with multiple embedded boards equipped with sensors and actuators. Each actuator or sensor controls a specific component of the machine (e.g., a spinning gear). Physical manipulation of these devices changes the state of the associated component in the holographic twin of the machine in real-time. For the user-study, we have the system starting in an unstable state, which means that one or more parameters are not set properly and need to be adjusted. Hence, the goal for the user is to bring the machine to a stable state by interacting with different components (e.g. align spinning gears, control their speed, avoid overheating). Users are then notified about the task completion through the interface they were using: an Head-Mounted Display (HMD) with holograms or a SCADA-like web-interface on a tablet. When using the HMD, the hologram changes in real-time accordingly to the user inputs. In contrast, the web interface only provides textual feedback. In the following, we report only the applica-

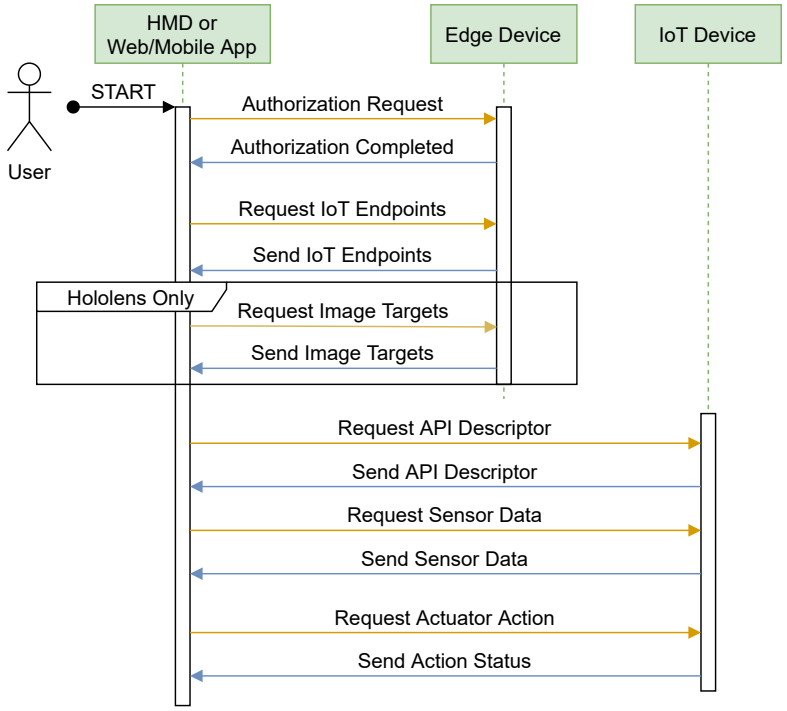


Figure 3.9: System workflow.

tion benchmarking results while the ones for the user-study can be found in the relative research paper attached to the thesis (Publication V).

For our experiments, we used the Microsoft Hololens (version one)[138]. Figure 3.10 shows the performance results of our application collected with the Windows Performance Recorder[4] and subsequently analysed with the Windows Performance Analyzer[5].

The CPU utilization shows peaks caused by the image recognition library, which includes the loading of recognition data paired with the IoT components discovery. Thus, based on the overall CPU load during the application usage, we conclude that the HoloLens has sufficient CPU power to perform image recognition tasks without negatively impacting the user experience. The Graphic Processing Unit (GPU) usage is heavily affected by the User Interface (UI) panel rendering, which also influences placement of and interaction with holograms. Frames per Second (FPS) were definitely sufficient to avoid motion sickness with an average of 48 and the usability testing showed that even with just 20 FPS (during complex holographic visualizations) the user experience was not compromised.

System power consumption represents the amount of power complexly used by the Hololens while SoC power consumption amounts only for CPU, GPU and memory. All values (except FPS) are represented as percentage. Power consumption was substantial during all our experiments as the application stressed particularly the embedded GPU,

[4] https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-recorder
[5] https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer

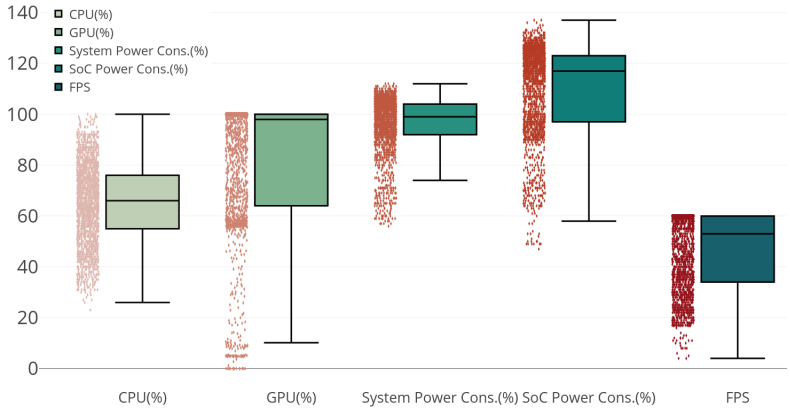


Figure 3.10: Application benchmark.

leading to a high System-on-a-Chip (SoC) power consumption value. Considering the device's autonomy of 113 minutes and its charging time of 1 hour, we conclude that battery life can be a serious problem in situations where prolonged utilization of the HMD is required. Hence, developers should pay great attention in designing their applications to avoid excessive power drain.

In our experiments, it emerged that the Hololens can overheat leading to unexpected shutdowns and discomfort for the user. We used a SeekThermal CompactXR[6] infrared camera to monitor the device temperature over time. After an average of 30 minutes, it reached a peak of 43.3° Celsius (our lab temperature was 29° Celsius) before shutting down effectively preventing any kind of interaction.

Edge computing can help address some of the problems emerged in our study. For example, overheating could be mitigated by offloading some parts of the computation such as rendering step in order to reduce the strain on the embedded GPU. The same holds for deep learning models required to recognize the machine to interact with its components. In this case, battery consumption could be potentially reduced — a topic that will be explored in more details later in Section 4.8.1.

3.2.3 Discussion

In this section, we presented a hologram-based framework for the manipulation and control of IoT devices in industrial settings. We built a prototype where a network of IoT devices connected to an edge gateway allows to manipulate the state of a virtual machine presented as an hologram to the end-user. From our evaluation it emerged that this AR applications are still in their infancy. In fact, our system benchmarks revealed the limitations of existing HMDs that can, however, be mitigated by leveraging the functionalities offered by edge computing. The presence of nearby devices offering extra computational power has multiple benefits. For example, it is possible to offload complex computational steps dramatically increasing multimedia applications performance in terms of latency.

[6] `https://www.thermal.com/`

Other benefits are extended battery life for the mobile device or support for old devices that would not be able to run such multimedia applications at all. We will come back on this topic at the end of Chapter 4, where edge-cloud resource provisioning challenges are discussed especially in the context of mobile AR applications.

3.3 Summary

In this chapter, we presented two domain-specific contributions leveraging the advantages provided by edge computing in different fields. UIDS can be seen as non-latency critical applications as its main selling point is its lightweightness and efficiency in running on constrained devices while not compromising on any security features. On the other hand, the virtual factory looks at the role of latency in the QoE of AR industrial applications while highlighting the advantage of using the support of an edge node in proximity. In this chapter, we looked at domain-specific problems that can be mitigated or solved with the support of edge computing. As a consequence, they are not application-agnostic and rather focused on specific use-cases. In the next chapter, we will take on a different set of challenges found at the intersection between edge computing and distributed systems. Hence, we will present the core contributions made by this thesis aimed at building a virtualized and distributed edge-cloud platform.

4 Building Blocks for Lightweight Edge Computing

Chapter 3 explored the potential of edge computing in two specific application contexts. Here, we focus on fundamental questions which are decoupled from a specific scenario or use-case. One aspect that is shared by the majority of research proposals and industry applications is the importance of resource management especially when serving many users. Network bandwidth, storage, compute capacity (CPU, GPU) and I/O are constantly managed in datacenters to decide how and where to execute services to maximize throughput. The process of managing resources it tightly coupled with the practice of *virtualization* which was introduced at first in the 1960s: a method of logically dividing the system resources provided by mainframe computers between different applications. [1] Over the years, virtualization has become a key component for the majority of the cloud providers. In fact, it provides the capability of pooling computing resources from clusters of servers and dynamically assigning or reassigning virtual resources to applications, on-demand [43]. However, it comes at a cost as virtualization technologies introduce another complexity level for the infrastructure provider, which has to manage both physical and virtualized resources [139]. In order to manage resources in a DC, extensive use is made of resource provisioning algorithms and tools such as Google Borg [140] and Apache Mesos [141]. Cloud DCs extensively use such tools to reactively [142, 143, 144, 145] or proactively [146, 147, 148] re-distribute the current load (e.g., by replacing or migrating VMs) in a datacenter to avoid potential performance degradation due to overload [149, 150]. Provisioning insufficient resources to customer applications can violate the Service Level Agreement (SLA). Algorithms behind VMs migration are only one of the facets of resource management challenges. In edge-cloud infrastructures, one of the main problems is deciding how and where to schedule tasks offloaded to the system by end-users (e.g., mobile devices) [151] in order to meet the application requirements. In this case, the research questions shift towards an user-centric perspective where the goal is to improve the delivered QoE.

To summarize, in this chapter we expand on the role of these two aspects in the edge computing domain. We look at ways of optimizing, adjusting and fine-tuning virtualization techniques and resource management algorithms for edge computing infrastructures. We will start by discussing how emerging virtualization techniques can be exploited at the edge in combination with the necessary orchestration platform support. Subse-

[1] As a matter of fact, there are two types of virtualization: virtualization of resources within the OS (e.g., processes) and virtualization of baremetal resources across multiple OSes. In this thesis, we focus on the latter.

quently, we tackle the problem of managing compute capacity at scale in a distributed, multi-tier, edge-cloud infrastructure.

4.1 Virtualization Techniques

In this section, we introduce the different available virtualization techniques which is a crucial step in later understanding the design choices embraced in our research work. We start with the origin of the system virtualization concept and progressively introduce the latest available techniques.

System virtualization has drastically evolved in the last years offering system architects and developers a vast array of techniques. The virtualization technique of choice for most open platforms over the past 5 years has been the Xen hypervisor [152]. However, there are other tools available such VMWare ESX [153], Oracle VirtualBox [154], Parallels Virtuozzo [155], OpenVZ [156]. Understanding how and when to utilize a specific technology based on the hardware constraints and application requirements becomes a crucial step in the system design phase. Today, many forms of virtualization are available and extensively used in datacenters to cater for the needs and requirements of large numbers of users and applications [157].

At the heart of a virtualized system there is the Hypervisor or Virtual Machine Monitor (VMM). Based on Popek and Goldberg's 1974 paper [158], there are three essential characteristics for a system software to be considered a VMM: *fidelity*, *performance* and *safety*. *Fidelity* assumes that the software on the VMM executes identically to its execution on hardware, barring timing effects. *Performance* requires that most guest instructions are executed by the hardware without the intervention of the VMM. *Safety* is achieved when the VMM manages all hardware resources so that it is impossible for an arbitrary program to affect the system resources, e.g. memory, available to it by bypassing the VMM. Classical virtualization was based on a technique called trap-and-emulate which, originally, was not supported by x86 architectures. As mentioned before, the VMM as complete control over any guest OS. In some cases, the VMM will intervene whenever one OS is attempting to do something that conflicts with what another OS wants to do. With the support of the CPU (VT-x instructions on x86 architectures), the OS is able to *trap* such attempts, and allow the VMM to *emulate* the effect that is desired by one OS, but in a manner that does not interfere with any other OS. With the introduction of binary translation (BT) and hardware extensions, it became possible to overcome such limitation and open to the virtualization technologies described in this section.

4.1.1 Classic Hypervisors and VMs

The physical resource partitioning is usually performed by hypervisors, which are software, firmware or hardware components that create and run virtual machines. They are classified in two types [158]: Type-1 and Type-2, as shown in Figure 4.1. The former, also known as *bare-metal* or *native*, run directly on the host's hardware to control the device resources and to manage guest operating systems. Xen [159] and VMware ESXi

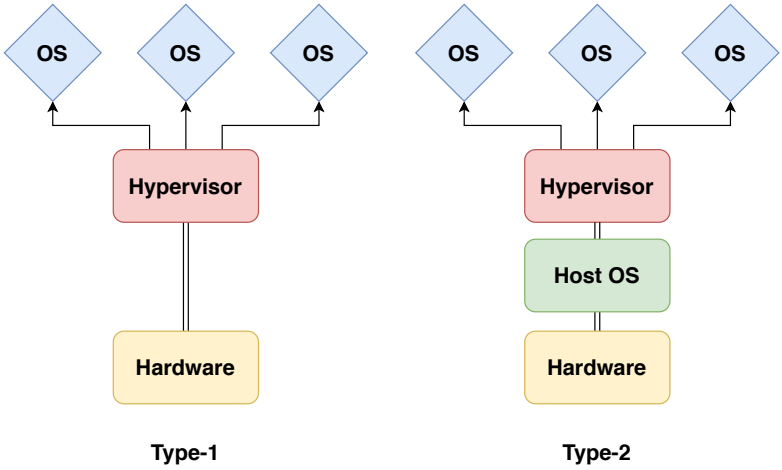


Figure 4.1: Comparison between Type-1 and Type-2 hypervisors.

[160] are examples of such hypervisors. Type-2 hypervisors instead run on top of conventional OS without directly taking control of the hardware resources. As a results, guest operating systems run as processes on the host. KVM [161] and QEMU [162] fall into this category.

Although the purpose of Type-1 and Type-2 hypervisors is identical, the presence of an underlying OS with Type-2 hypervisors introduces unavoidable latency as all of the hypervisor's operations and the work of every VM pass through the host OS. Also, security flaws or vulnerabilities in the host OS could potentially compromise all of the VMs running above it. Therefore, Type-2 hypervisors are generally not used in DCs and are reserved for client or end-user systems – sometimes called client hypervisors – where performance and security are lesser concerns.

Depending on the type of emulation provided by the hypervisor, the hardware virtualization can be divided into *full-virtualization*, *para-virtualization*, and *hardware-assisted*:

- **Full-virtualization.** In this case, the virtual hardware exposed is functionally identical to the underlying machine [163]. Full-virtualization is usually based on binary translation which allows to detect and replace unsafe or privileged instructions (e.g., I/O operations) which can affect the state of other VMs or the underlying hardware. No modifications of the guest OS are required, such as host interfaces like para-API. Full-virtualization entails a considerable performance degradation in comparison to natively executed systems [159]. This overhead comes from the need for the hypervisor to fully simulate the actual hardware, which noticeably increases the complexity of the VMM. However, there are cases in which it is desirable for the hosted operating systems to see real as well as virtual resources. For example, providing both real and virtual time allows a guest OS to better support time-sensitive tasks, and to correctly handle TCP timeouts and RTT estimates. Exposing real machine addresses allows a guest OS to improve performance by using superpages [164] or page coloring [165].

- **Para-virtualization.** It is a technique which simulates a virtual hardware platform and enables the execution of virtualized guest OSes. Para-virtualization is necessary to obtain high performance and strong resource isolation [159]. Since the hardware is simulated, the guest OS requires modifications in order to be compatible with the physical hardware [166]. Specifically, so called *front-end* interfaces (or para-APIs) translate commands from the actual hardware to the simulated one. Such interfaces reduce the complexity of the hypervisor by moving the execution of some complex operations to the host domain. However, there is no need to apply any changes to the Application Binary Interface (ABI), and hence no modifications are required to guest *applications*. For example, the MirageOS unikernel, which we used extensively in our work, only knows how to create para-virtualized Xen images when compiled to run as a virtual process on top of the Xen backend.

- **Hardware-assisted Virtualization.** Today, modern CPUs offer hardware inbuilt virtualization support to boost virtualization performance. Examples include Intel Virtualization Technology (VT) [167] and ARM Virtualization Extensions [168]. Hardware-assisted virtualization reduces the maintenance overhead of para-virtualization as it reduces the changes needed in the guest operating system. It is also considerably easier to obtain better performance. This depends primarily on the frequency of *exits* of the guest, which are triggered, for example, in case of I/O operations. VM exits in response to certain instructions and events (e.g., page fault) mark the point at which a transition is made between the VM currently running and the VMM. A guest that never exits and only computes, can run at native speed. However, this is seldom the case for standard VMs. The exit rate is a function of guest behavior, hardware design, and VMM software design: a guest that only computes never needs to exit; hardware provides means for throttling some exit types; and VMM design choices, particularly the use of traces and hidden page faults, directly impact the exit rate [158]. Therefore, it was important to reduce the frequency of exits calls with the first generation of hardware-assisted virtualization in order to maximize its usefulness and performance. With time, virtualization techniques have evolved and introduced new features such as the input/output memory management unit (IOMMU) which helps in reducing the communication overhead with peripherals including Ethernet, GPU, and HDD controllers (also known as Peripheral Component Interconnect (PCI) passthrough [169]).

In the next section, we will move on to a more recent form of virtualization which is currently heavily used by many cloud management tools such as Kubernetes [170] and is generally known as *containerization*.

4.1.2 OS-level Virtualization and Containers

Differently from standard hypervisors, *OS-level virtualization* is an operating system paradigm in which the kernel allows the existence of multiple isolated userspace instances. These run and rely upon the underlying host system meaning that every single one also shares the same host kernel through the virtualization engine. Instead of virtualizing all

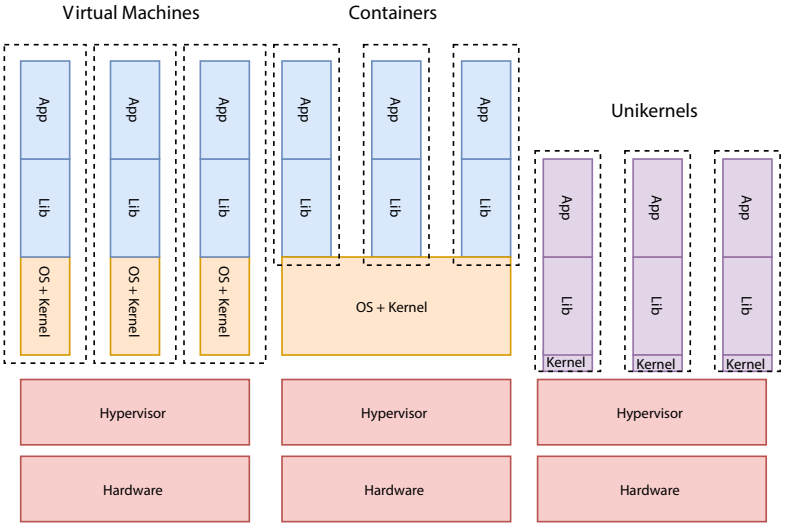


Figure 4.2: Comparison of different virtualization technologies.

the physical hardware, only the software stack sitting on the kernel is virtualized. As a consequence, OS-level virtualization offers little to no overhead as the containers are running directly on top of a shared kernel. This form of virtualization is often called *containers* of which some popular implementations are Docker [171] and LXC [172]. While VMs encapsulate the entire state of a running system, including both user-level applications and kernel mode operating system services, containers provide operating system services from the underlying host and isolate the applications using virtual-memory hardware. In a nutshell, a VM provides an abstract machine that uses device drivers targeting the host machine, while a container provides an abstract OS [173].

4.1.3 Library Operating Systems and Unikernels

Radical OS architectures from the 1990s introduced the concept of *library operating system* (libOS). In a libOS, protection boundaries are pushed towards the lowest hardware layers, providing: (*i*) an ensemble of libraries to interact with hardware or network protocols, and (*ii*) rulesets to set protection boundaries for the application layer. Some examples are Exokernel [119] and Nemesis [120]. A few advantages of a libOS are increased performance due to reduced context switch between user and kernel space, tiny attack surface compared to VMs and containers, fast boot-up time, and extremely small footprint with only around 4% the size of the equivalent code bases using a traditional OS [174]. However, libOS also have drawbacks among which the biggest ones are additional complexity in isolating multiple applications resources and the cost of rewriting device drivers to fit the new architecture. The advent of hypervisors helped mitigating these issues and opened the doors to unikernels. In particular, para-virtualization brought to unikernels virtualized disk and network drivers, interrupts and timers, emulated motherboard and legacy boot, and privileged instructions and page tables.

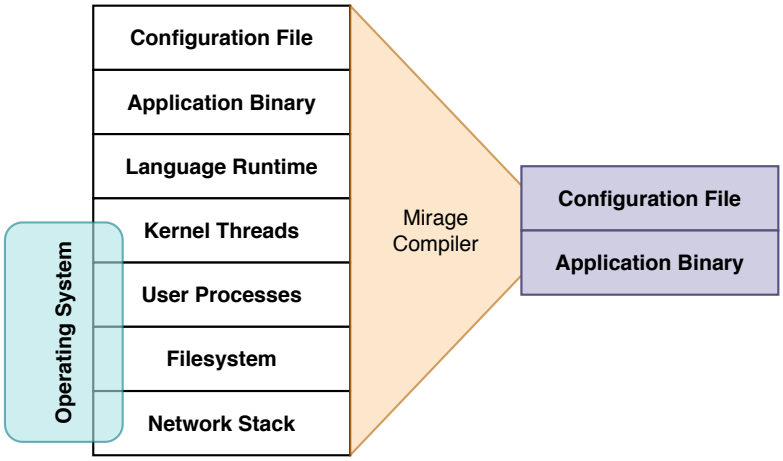


Figure 4.3: MirageOS unikernel generation process.

Unikernels are single-purpose appliances that are at compile time specialized into standalone kernels [122], and sealed against modification after deployment. They are written in a high-level language and act as individual software components. Generally, a full application consists of a set of running unikernels working together as a distributed system [175]. Figure 4.2 shows the differences between the virtualization technologies discussed before and unikernels.

Unikernels can be seen as an extreme form of lightweight virtualization emerged from the observation that most virtualized applications are often burdened by additional, unnecessary functionalities and libraries inherited from the underlying OS. For example, VMs have many software layers while, ultimately, performing a single function including a database or Web service. This represents a real opportunity for optimization both in terms of performance, by adapting the virtual instance to its task, but also for improving security by eliminating needless functionality. Their attack surface is strictly confined to the running application logic as everything that is part of the unikernel is directly compiled into the application layer. Therefore, each unikernel may have a different set of vulnerabilities, implying that an exploit that can penetrate one may not be threatening the others. However, the high degree of specialization means that unikernels are unsuitable for general purpose applications. Adding functionality or editing a compiled unikernel is generally not possible, and instead the approach is to compile and deploy a new unikernel with the desired changes.

Initially designed for public clouds, unikernels are also potential virtualization candidates for edge-cloud networks due to their small footprint and flexibility, as shown in other research efforts [176]. Unikernels have been primarily designed to be stateless, similarly to lambda functions. Therefore, they are a good fit for standard stateless functional algorithms [177] or for Network Function Virtualization (NFV) [178]. There are multiple available unikernel implementations which differ mainly in the supported programming language. MirageOS [122] (architecture outlined in Figure 4.3) is a unikernel based on the OCaml functional language which aims at unifying both

kernel and application userspace into a single, high-level framework. Among other benefits this brings static type-checking, automatic memory management, modularity and meta-programming (optimizing compiled code based on runtime parameters). Similarly, HaLVM [179] is a unikernel based on Haskel with pervasive type-safety in the running code. IncludeOS [127] and ClickOS [180] support C++ with the former coming in the form of a framework to which is possible to bind any application. The latter is highly specialized in offering functions to do network traffic processing (based on the Click modular router [181]). OSv [182] is a Java based unikernel offering more flexibility at the cost of additional overhead introduced by the Java Virtual Machine (JVM). Unikraft [183] is another micro-library OS that fully modularizes OS primitives so that it is easy to customize the unikernel and include only relevant components. Additionally, it exposes a set of composable, performance-oriented APIs in order to make it easy for developers to obtain high performance.

Unikernels are good candidates for the creation of systems based on microservices and serverless architectures. This is especially true at the edge, where we can benefit the most from the unikernels perks such as small memory footprint and reduced attack surface. In this context, lambda functions are also a promising option which will be described in the next section.

4.1.4 Lambda Functions

In the past 15 years, many serverless services have emerged such as Google App Engine[2], AWS Lambda[3], Kubeless [184], and OpenLambda [185]. In the cloud context, serverless meant that developers should not worry about servers and in particular just uses SaaS platforms or services [186]. This means that a developer can focus on writing code without having to manage underlying infrastructure or worry about challenges such application scaling or security [187]. The latest serverless solutions are server-hidden and built to host functions that may be part of a pre-existing service (e.g., Google Cloud Datalab[4]) or offered as an independent service in a *Function as a service* (FaaS) fashion. In particular, FaaS platforms based on lambda functions have received increasingly more attention due to the simplicity with which code can be deployed into production. The principle is to offer a compute runtime where stateless, non-virtualized, functions are executed. The challenge is to design such a system while considering metrics such as cost, scalability, and fault tolerance [188]. Functions must be rapidly booted to process their input. The system also needs to queue events and based on them schedule the execution of functions, and manage stopping and deallocating resources for idle function instances. In addition, the system has to handle failures in a cloud environment, at scale. The serverless computing paradigm can excel in short-running, stateless, and event-driven systems. However, it is not the best fit for complex, long-running, and stateful computations or real-time demands.

[2] `https://cloud.google.com/appengine`
[3] `https://aws.amazon.com/lambda/`
[4] `https://cloud.google.com/datalab/docs`

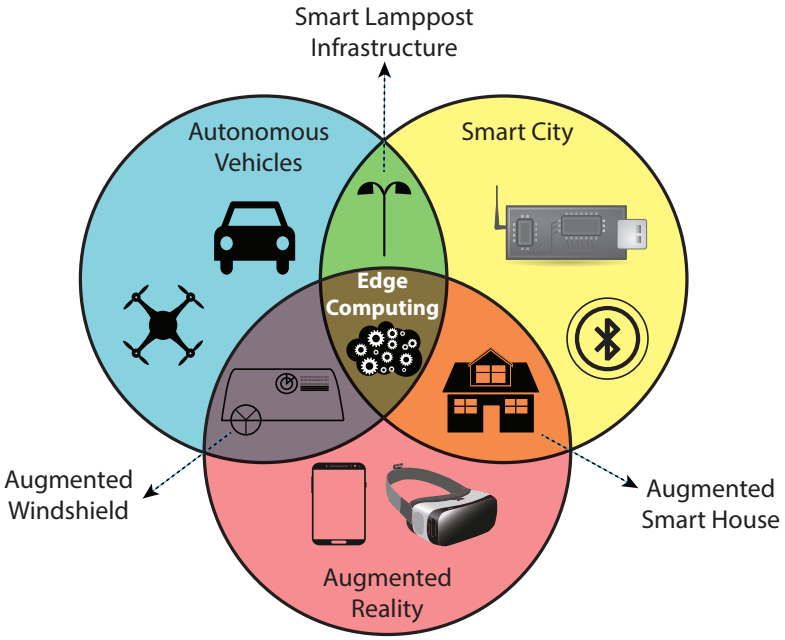


Figure 4.4: Application scenarios enabled by LV and the edge-cloud infrastructure.

There are many similarities between lambda functions and unikernels, especially in terms of orchestration requirements from the underlying system managing them. Nevertheless, we will not discuss them further and, from now on, we will focus on virtualization-based solutions.

4.2 Virtualization for Edge-Cloud Computing

Virtualization is core enabler technology for the interplay between cloud and edge by offering an abstraction layer where resource-contrained devices and servers can coexist in an unified infrastructure. In our approach we focus on what we call Lightweight Virtualization (LV) and, in particular, unikernels which are a promising candidate to address the manifold challenges found at the edge and in the IoT domain such as security, scalability, hardware heterogeneity, resource-constrained devices. LV includes and represents all forms of virtualization that do not rely on heavy-weight, full-blown VMs but rather slim, compact virtualized applications using a bare-minimum OS stack. This was also shown previously in Figure 4.2.

A direct benefit emerging from employing LV at the edge is flexibility. Within a lightweight virtualized instance (e.g., container, unikernel) we can efficiently deploy an application able to manage and use different technologies offered by the host device. Equipping edge devices with newer services becomes easier, since we only need to configure and instantiate stand-alone virtualized applications. In particular, referring to the edge-cloud architecture discussed in Chapter 2, it is crucial to provide simple and yet

efficient instantiation methods that are technology-independent in terms of hardware capabilities and cloud provider. The tools used at the edge layer should share common functionalities and exploit similar APIs to orchestrate and interconnect different networking and computing technologies. However, services and application might have different requirements such as scalability, multi-tenancy, privacy & security, latency, and extensibility.

Compared to classic virtualization solutions, we envision a trend towards using LV technologies in an edge-cloud infrastructure. A direct benefit emerging from employing LV at the edge is flexibility. Within a lightweight virtualized instance (e.g., container, unikernel) we can efficiently deploy an application able to connect and use different technologies offered by the host device. In addition, equipping edge devices with newer services becomes easier, since we only need to configure and instantiate stand-alone virtualized applications. Complex re-programming and updating operations that are part of the software lifecycle management are, therefore, avoided. In fact, updating a particular service requires changes only within a specific virtualized instance. LV can also enable cross-platform deployment, providing a common execution environment across cloud, edge, and constrained IoT devices. This would allow the whole infrastructure *to speak the same language* and adapt to each layer capabilities and requirements. Based on these, a service deployed on an edge-cloud infrastructure is split into a set functions which are then deployed on the different devices based on the their capabilities. This lays the foundations for a decentralized edge-cloud service provisioning architecture where tasks are performed in a cooperative fashion.

Figure 4.4 gives on overview of possible application scenarios that can be enabled by LV in conjunction with and edge-cloud infrastructure. We identified three main areas: autonomous vehicles, smart city, and augmented reality. In the following, we will focus on two selected use-cases (one latency and one non-latency sensitive): the smart infrastructure and real-time applications, respectively. We will discuss the reasons for adopting a specific LV technology for both cases. Specifically, we focus on unikernels as an essential element of this thesis.

4.2.1 LV for Smart Infrastructure

Over the last decade, the development of the Internet of Things has been fueled by the cloud-based infrastructures that aim to cope with the increasing number of IoT services provided by various connected devices. This has (obviously) generated an intrinsic association between IoT and cloud, where the cloud-based network infrastructures are optimized to support a multitude of IoT-centric operations such as service management, computation offloading, data storage, and offline analysis of data. This model has its limits as already discussed in Chapter 1 and 2.

One problem that edge computing can help addressing is hardware reusability. In fact, different cloud providers will aim at deploying their own, customized sensors which will be only able to talk to a specific service. This creates a considerable amount of redundancy in terms of deployed hardware and waste of resources. Moreover, this fuels the problem of *e-waste* [189] which is considered the fastest-growing waste stream in the

world [190]. With edge computing, this problem can be addressed by introducing an intermediate management layer between cloud and IoT where different services can be scheduled allowing the re-use of sensors for different applications. The edge layer will take care of managing the hardware resources which become agnostic to the business logic running in the cloud. Additionally, by offloading part of the application logic at the edge, with this model we reduce the amount of data that needs to be uploaded to the cloud and leverage local compute capabilities. There are multiple examples where this solution can be applied such as environmental data monitoring, smart lighting, smart grid, and traffic management [191]. For example, this is especially fruitful in situations where the infrastructure is deployed by the public administration and lent to private cloud service providers based on specific SLAs.

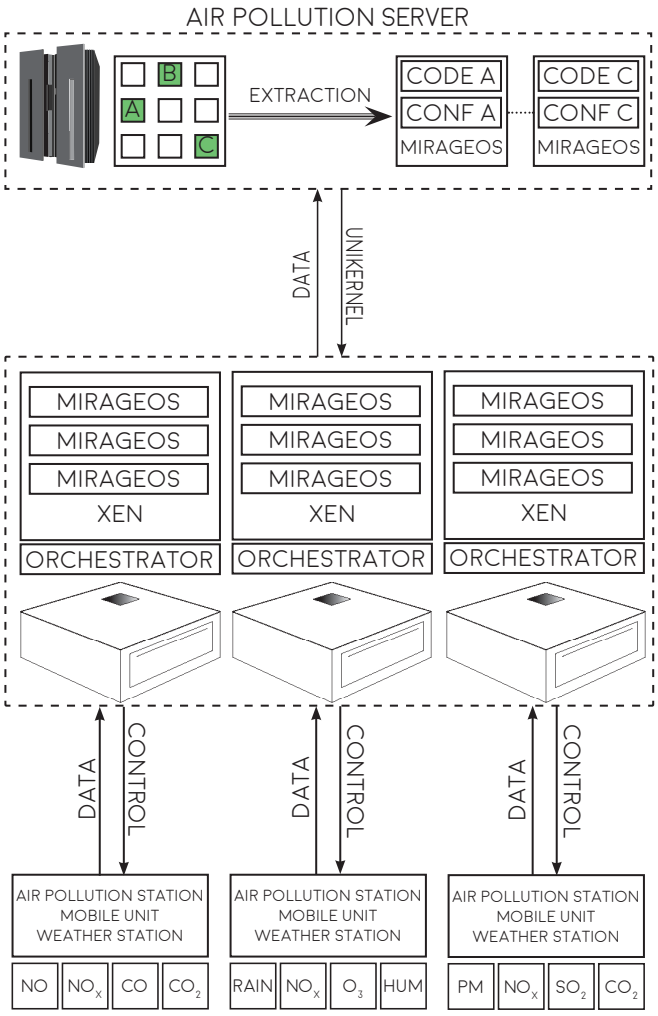


Figure 4.5: Air pollution monitoring scenario.

An example for which we provide additional details is the measurement of environmental data which has become an crucial issue especially for densely populated metropolises. Currently, air pollution monitoring is achieved with a sparse deployment of stationary, expensive measurement units embedding both sensors[5] and computing units. Air pollution is predicted based on the measured data in combination with complex mathematical models [192]. As the deployment and maintenance cost of such such pollution station is often prohibitive, we envision crowd-sensing as a tangible solution that combines LV and edge computing.

The edge computing layer offers resources close to the crowd-sensing entities, which can offload their data to the edge node in proximity which will take care of processing them locally. In relation to the virtualization technologies discussed above, unikernels can be used to take care of the computation with multiple of them spawned on-demand. Each one would contain only the code necessary to process a subset of the received data as a function, for example, of the sensor data type or to serve a particular application. The partial results will be then subsequently uploaded to a more powerful edge node (e.g. edge micro-server) to be merged or receive additional processing. Some of the benefits of this approach are reduction of the load on the core network and of the cloud (and air pollution stations) provisioning costs. In this case, unikernels are a good match as the algorithms used to assess air pollution levels are generally static and stateless. In other words, they can be considered as black-boxes with a defined set of inputs and outputs. Figure 4.5 shows an example architecture in line with what we described. In particular, while the bottom part of the figure is specific to use-case described above, the middle and top part are general purpose. To be more precise, this architecture can be applied also to other scenarios and extended to support other services.

4.2.2 Real-time and Multimedia Applications

Since the advent of consumer mobile devices equipped with multiple sensors and powerful chipsets, multimedia applications have garnered increasing interest amongst smartphone users. Currently, mobile AR adoption stands at 32% with 54% of its users using it at last once a week [100]. Despite the increasing popularity of the technology, mobile AR applications often offer substandard QoS and user QoE. The reason for this is two-fold. Firstly, AR depends on complex deep-learning algorithms which are a bottleneck [193] as the front-end devices are often not powerful enough to execute them with acceptable latency for the end user [194]. Secondly, extended usage of multimedia applications results in a considerable power consumption, which leads to significant battery drain and overheating [195, 196]. One solution to the problem is offloading AR tasks to cloud backends in order to cut on the device power consumption and compensate with potentially insufficient the mobile device processing capabilities. However, cloud services introduce additional latency, which negatively affects real-time applications especially in terms of user immersion and motion-to-photon delay. For example, a study revealed the

[5] Usually gas detection sensors (NO, NOx , O3, CO, CO2 and particulate matter) plus humidity, rain detection and wind speed/direction.

requirements of virtual reality applications to achieve perpetual stability [197]. Longer delays in such highly interactive, real-time applications degrade the end-user experience.

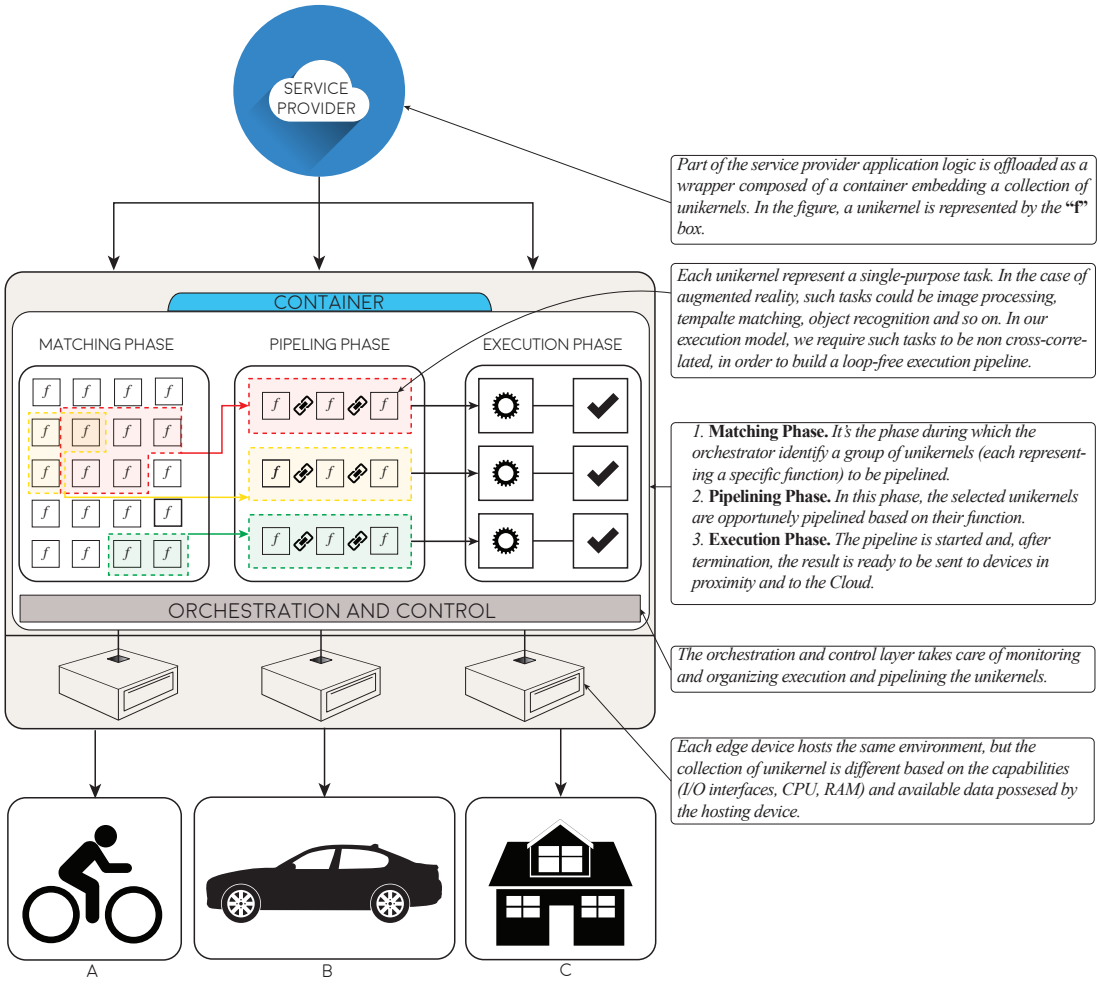


Figure 4.6: Real-time application scenario. (A) Cyclist receiving personalized advertisements rendered in AR on their smart glasses; (B) A smart car populating its augmented windshield with contextualized, live feed information; (C). Augmented smart home, where we control IoT devices in proximity through virtual interfaces.

Referring to Figure 4.4, one interesting application is personalized infotainment for mobile users such as bike riders or smart vehicles. Information collected and processed by the roadside infrastructure are sent to the road users which can use them to enrich their mixed reality experience. For instance, such information could be traffic conditions, personal agenda, news feed, gaming interfaces, social networks and so forth. In order to craft and manage such a visually-rich experience, an edge board mounted on the car is considered necessary. Based on the capabilities of the mobile device, the edge-cloud network can provide different levels of support. Powerful mobile devices might not immediately need to offload computer vision tasks and require only preprocessed

information from the smart infrastructure. Low-end clients, instead, might want to offload some computationally expensive computer vision steps of the AR pipeline to edge nodes in proximity. The use of virtualization in this context is motivated by the need to multiplex and manage discrete resources (e.g., embedded GPU) across different services and users. Additionally, isolation and privacy are important to preserve as sensitive, user data might be used in some parts of the processing steps.

Another use-case highly dependent on latency is cloud gaming [198]. This is based on streaming a live feed of the game directly to a device while the game itself is processed and running in a datacenter. Cloud gaming companies aim at building edge servers as close to gamers as possible in order to reduce latency and provide a fully responsive and immersive gaming experience. Additionally, the edge enables new cloud gaming platforms to eliminate the need of dedicated devices, such as a console or high-end personal computer, while helping solve the latency issues in transferring data from the cloud to the user and the rendering of graphically intensive video [199, 200, 201]. The scenarios mentioned above call for different functions to be executed and work together in the form of a pipeline. As shown in Figure 4.6, we consider the combination of containers (e.g., Docker) and unikernels as a potential approach. In fact, a container embedding multiple unikernels can be built and shipped, where each one of the latter contains either data or image processing operations. In this case, the container would offer orchestration and control APIs to external applications with, under the hood, unikernels embedding the application logic.

4.3 Deployment and Management

In our idea of a distributed edge-cloud infrastructure, device orchestration becomes a crucial problem to tackle. Without orchestration and synchronization, it is not possible to have different devices cooperate to solve a task. Additionally, specific tools are required to deal with different processor architectures, storage capacity and network configurations of edge devices and cloud DCs. Making the best use of the edge infrastructure calls for proper knowledge of the available hardware resources in order maximize their utilization without overloading the devices at the edge. This calls for a lean orchestration framework supporting LV technologies seeking a fair balance between synchronization and network load. Other key aspects concern the definition of optimized policies for an efficient *vertical scaling*, in which applications are automatically prioritized and scaled up/down in the edge-cloud infrastructure, according to specific QoS requirements or eventual computing resources saturation at the edge.

Mobility is also a relevant aspect. User devices might move in relation to the edge-processing node providing the service. Therefore, the computation may need to be re-deployed or migrated multiple times at different locations to transparently serve mobile users. This is challenging especially for custom, stateful services bound to individual users which require also application state migration or synchronization. In this case, for cloud-native service an option is to avoid storing state locally or only to use soft state. For services requiring local state, there is the possibility to store the current state at

an external stable location before exiting and load it again on restart. Consequently, particular attention must be paid to ensure that destination edge device has enough available resources to run the service. Otherwise, an alternative candidate should found to run the service to avoid resource overbooking.

4.4 Orchestrating Unikernels

With unikernels, application are treated as libraries within a single application, allowing the developer to configure them using either simple library calls for dynamic parameters or meta-programming tools for static parameters [175]. The result is a reduction in the effort needed to configure complex multi-service application VMs. Currently, there are no publicly available unikernel orchestration tools and hypervisors are lagging behind in evolving towards the structured unikernel worldview. In our work, we progress in the direction of an agile set of intercommunicating, tiny virtualized instances that can be scheduled and restarted independently, in the spirit of a distributed system of micro-services.

We dedicated part of the research effort in this thesis to designing and implementing the first unikernel orchestration framework compatible with both resources-constrained devices and cloud servers. Specifically, expanding on the previously shown image describing our methodology (Figure 1.2), we introduce the three fundamentals block explored in the first half of this chapter as shown in Figure 4.7:

- **FADES (Orchestration).** An edge-cloud offloading architecture allowing to run MirageOS unikernels to support a variety of IoT and cloud services. The design principle behind FADES is to efficiently exploit the resources of constrained edge devices through fine-grained computation offloading.
- **ECCO (Chaining, Distributed Computing).** This evolution of FADES enables edge-cloud collaborative computing through on-demand task execution pipelines spanning multiple, potentially resource-constrained edge-nodes.
- **MirageManager (Migration).** To address the problem of stateful applications migration, we developed an unikernel migration system enabling lossless migration supported by a function-level, application logic checkpointing library of our design.

4.5 FADES

Simplicity is key to the Internet of Things. Regardless of back-end services, resource-constrained edge devices should execute simple operations on data locally available in order to maximize efficiency. Therefore, by splitting a complex application into manifold simple and single-purpose tasks we can ship them in the shape of lightweight containers (specifically, unikernels). This approach follows the best-practices of microservices based

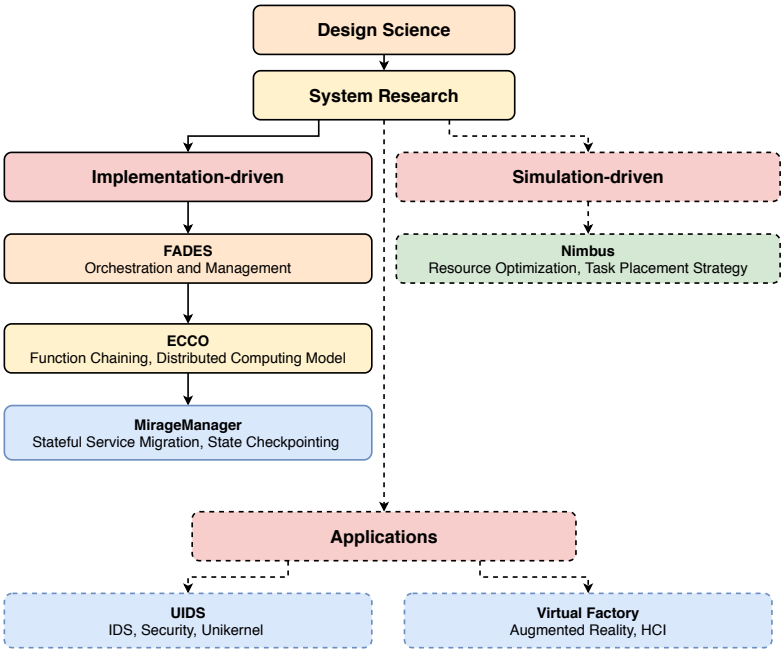


Figure 4.7: Contributions: implementation-driven branch.

architecture where developed components are re-used in multiple systems in a plug-and-play fashion.

IoT diversity is a double-edged sword. In fact, on one side we face the problem of heterogeneity and lack of standards. On the other, the massive scale of devices is can prove to be a powerful source of information and computational power. The core motive behind FADES is exactly to make use of this power in an efficient way. As a matter of fact, the trifecta of locally available resources, computational power and hardware capabilities is the key metric to drive the task offloading process from the cloud to the edge.

In this section, we build on top of these observations and present FADES: the first prototype of our unikernel orchestration framework. We introduce the design and implementation of our system and then discuss some of the limitations that we addressed in the final version.

4.5.1 System Design

In order to run, orchestrate, and manage unikernels on edge devices, we designed and implemented from scratch a framework called FADES (Function virtulizAtion basED System). It is a modular system offering reliability, scalability and flexibility by leveraging MirageOS unikernels to embed application logic fragments (e.g., similarly to micro-services) in small, Xen-bootable images.

One of the core observations driving our system design is *data locality*: a principle based on which computation should gravitate towards the data source in order to cut

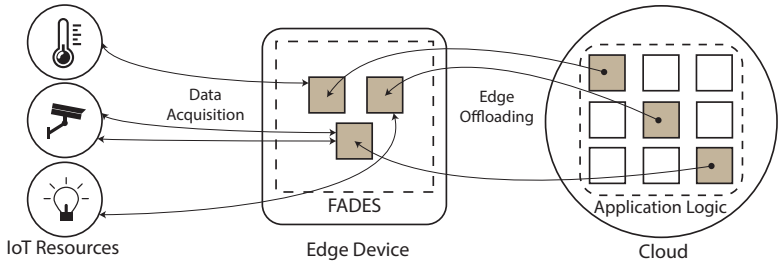


Figure 4.8: Edge offloading with FADES.

data transmission overheads. As edge devices have limited resources, they can only execute simpler tasks on locally available data compared to cloud services. Therefore, an application can be split into manifold simple and single-purpose tasks which can be shipped to the edge in the shape of lightweight virtual instances (specifically, unikernels). For example, we can offload at the edge parts of the application logic which need to access or process raw data generated by sensors, as shown in Figure 4.8. As a consequence, FADES takes advantage of edge devices as an intermediate layer where to offload selected stages of a compute flow with the goal of redistributing computation across the edge-cloud infrastructure. This multi-stage data and compute pipeline is also motivated by the necessity of reducing the uplink access parallelism. By offloading computation, we progressively aggregate data along the path to the cloud which also reduces the amount of data sent to the cloud, especially at scale. However, not everything can be offloaded. Application complexity, priority, criticality, power consumption and required physical resources play a key role in determining what can be offloaded and on which device. For example, applications requiring considerable computational power such as data mining or the learning phase of deep learning algortihms are better hosted in DCs rather than moved at the edge.

Figure 4.9 shows the system design of FADES. The entities on the left side of the figure are sensors, actuators, or other data sources from which raw information is collected (or to which commands sent, in the case of actuators). The External Services (ESes) are back-end applications interacting with the FADES framework which offload parts of their application logic in the form of unikernels. These ES can be seen as a repository of deployment-ready tasks designed for different scenarios and purposes. The main components of FADES include the Orchestrator (ORC), the Data Resource Broker (DRB) and Data Manipulation Functions (DMF). FADES is an event-based system using so called Metadata Task Wrapper (MTW) issued by the ES to exchange information about the offloaded task. An MTW is composed of three parts:

MTW Credentials contain passport-like information (e.g., task ID, associated user or service, priority) and are mainly used to keep track of the received MTWs and schedule their execution.

DRB Metadata are a list of *Data Retrieval Operations* (DTO) containing details about what data to retrieve, and these source and destination.

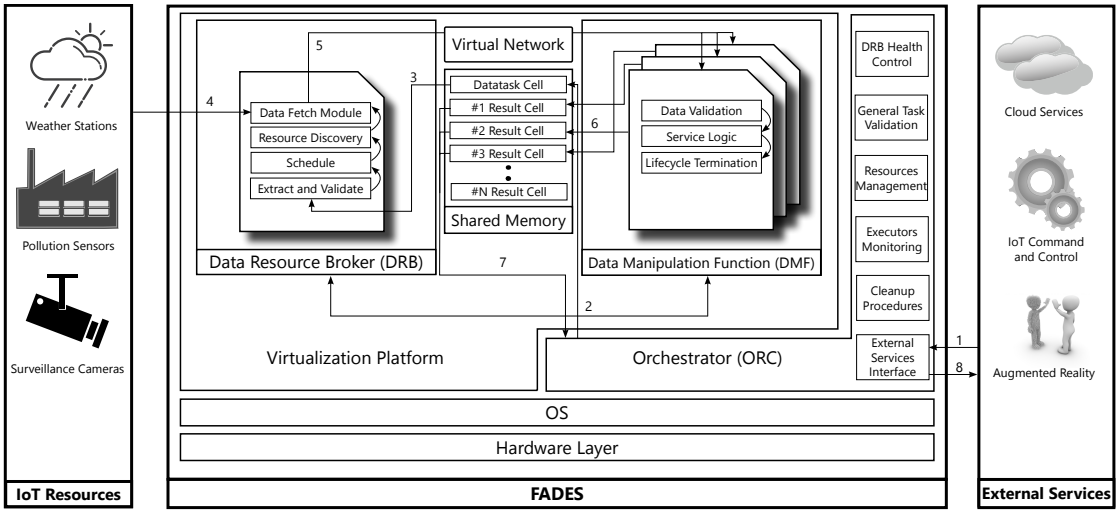


Figure 4.9: FADES design.

DMF Metadata provides application-specific details about the DMF. For instance, the MTW issuer can specify additional information about eventual extra runtime configuration parameters or minimal required hardware resources and capabilities.

The main components of our framework shown in Figure 4.9 are listed below:

- **Data Resource Broker (DRB).** This module localizes and extracts resources from the registered datasources which, in turn, will exchange with the DRB details about the available data and the policies to access them. The DRB is completely computation-agnostic. Its execution cycle is event-based and driven by the commands received by the orchestration module.

- **Data Manipulation Functions (DMF).** It embeds the application logic relative to a specific service and it is booted only when deemed necessary by the orchestrator. For example, when a specific event occurs (e.g., reception of new data from a sensor) or after an interval of time in case of DMFs scheduled to run periodically. DMFs can be persistent or ephemeral, based on their life-cycle. Long running or recurring applications might run in the form of daemons that only exit when meeting specific conditions. Single-execution tasks, such as ephemeral data aggregation procedures, are deallocated immediately after completion.

- **Orchestrator (ORC).** Is the interface between our system and the outside world. With a supervisory role, it monitors and controls the system by periodically checking that both the DRB and DMF are running correctly. The ORC also takes care of forwarding the required information to the DRB when a new MTW arrives. It ensures the overall system integrity by monitoring the DRB, following the life-cycle of each DMF, validating uploaded tasks and decommissioning terminated DMFs.

Recalling the air pollution monitoring example shown previously in this Chapter (Figure 4.5), we can map FADES components to the following tasks: (i) the ORC receives

from cloud the tasks to be executed plus additional metadata with the requirements its execution (e.g., type of sensor to access, CPU and RAM requirements), (*ii*) the DRB locates the accessible pollution sensors (or weather stations) to be queried and retrieves the data, (*iii*) the DMF manipulates the raw data received from the DRB and produces an aggregated results such as the level of pollution in a specific part of the city.

4.5.2 Implementation

FADES is developed as a virtualized, unikernel-based system running on top of the Xen hypervisor. Our tool of choice is the MirageOS library operating system, which is specifically designed to build modular systems and to run natively on Xen. Both the DRB and DMFs were embedded into MirageOS unikernels compiled against Xen as Para-Virtualized Machines (PVM). The DRB is implemented as a daemon unikernel, and to communicate with other components of our architecture, it uses two Xen modules: the virtual network stack and the XenStore. The former is used to internally exchange data with the DMFs while the latter is exclusively used to exchange synchronization messages with the ORC.

The DRB validates and schedules each MTW received from the ORC following different scheduling policies (e.g., sorting by task priority, execution deadline, task ID). Each DMF can execute a different operations (e.g., sensor fusion, image processing) and it is possible to run multiple of them in parallel. Once the result of the computation is ready, the DMF sends to the ORC the processed data through the Xenstore.

The ORC is implemented in Python and it is the only non-virtualized module in FADES. Moreover, the ORC is the only module that handles reads and writes towards the persistent storage. Our design choice focused on establishing a loose coupling between the host system and the unikernels managed by FADES, with the latter being exclusively dependent on virtual resources such as CPU, RAM and network. By following this practice, our goal is to minimize the assumptions and dependencies our system introduces in relation to the hosting device and in terms of libraries, OS, and special hardware capabilities.

4.5.3 Evaluation

Being our initial prototype, with FADES[6] we aim at understanding the applicability of unikernels at the edge where one of the main drawbacks is the presence of resources-constrained devices. Specifically, our goal is to produce a set of baseline benchmarks to understand the performance of our system. For the measurements, we used three devices: Cubietruck (ARM), Intel NUC, and a Dell PowerEdge R520 server. Additional details about the hardware capabilities and conducted experiments can be found in the research paper attached to this thesis (Publication II).

- **Memory Analysis.** When offloading to resource-constrained devices, it is important to make efficient use of the already limited available hardware resources.

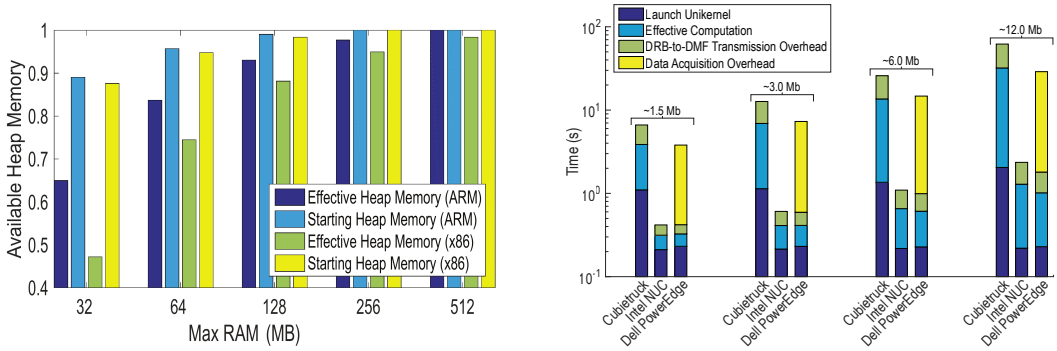
[6]Open-source code available at: `https://gitlab.lrz.de/vit.cozzolino/mirageos-iot`

We analyse how the available RAM affects the DMF performance. The goal is to avoid over/under dimensioning issues that could lead to waste of resources or memory depletion. respectively. Figure 4.10a shows the ratio between available heap memory and pre-allocated RAM with different architectures (x86 and ARM). On both architectures, the effective available memory is less than the amount allocated at the beginning but the behavior varies between x86 and ARM. In the first case, the gap is much more prominent, and this directly affects the amount of data processable by the unikernel especially in the case of a PVM with low allocated RAM.

Two main factors influence the available memory for a unikernel PVM: underlying architecture and imported libraries. The latter is a function of the specific application logic embedded in the unikernel. Therefore, it is a responsibility of the developer to avoid unnecessary tools or libraries. For the former, we noticed how different system architectures influence the available runtime memory. In fact, on ARM processors the output of the build process is a Linux kernel ARM boot executable zImage (.xen) together with an ELF 32-bit LSB executable (.elf) while on x86 only a single .elf file is generated. The difference between the the two build processes influences the size of the generated Xen image and affects the available memory at runtime, as shown in Figure 4.10a.

- **System Analysis: Overhead and Offloading.** We compare the performance of running a task at the edge, where the required raw data is directly available, against the cloud. Figure 4.10b shows the results with different payload sizes and providing a detailed breakdown of the execution time of a task in FADES. Each payload contained a times-series of varying length measurements obtained from our internal testbed composed of multiple Intel Edison[7] boards deployed in different office rooms in our university campus. Four main factors affect the overall execution time: *(i)* required time to boot the DMF, *(ii)* time required to transfer the data between the DRB and the DMF, *(iii)* time to run the computation logic in the DMF, and *(iv)* the time required to retrieve the data to be processed. The latter is only present for the cloud, which has an additional cost (in terms of latency) to retrieve the data from the edge. The bars in the figure are grouped by the amount of data to be processed. The processing executed inside the unikernel was a simple sensor fusion algorithm aimed at correlating raw sensors data (e.g., temperature, humidity, light intensity) to human behaviors/actions. Specifically, we embedded simple manipulation and aggregation functions (e.g., calculate minimum, maximum, average) over sensors data streams in the DMFs. Details about the source of the used data can be found in the respective research paper (Publication II). The results show that the presence of a sufficiently powerful edge device can complete the given task faster than the cloud, if the computation is moved closer to the datasource. In fact, while the Dell PowerEdge outperforms the edge devices in terms of computation time due to its faster CPU, it pays heavily in

[7] `https://en.wikipedia.org/wiki/Intel_Edison`



(a) Memory allocation: Heap RAM as function of max RAM.

(b) System performance.

Figure 4.10: FADES benchmarks.

terms of data transmission time. As the amount of data to be retrieved increases, the more this time penalty grows. Considering that often edge and IoT networks have limited uplink bandwidth and are unreliable, we consider offloading computation at the edge a valid solution to reduce overall task execution time, uplink bandwidth utilization, and the need for retransmissions due to eventual network instabilities. More details regarding the network bandwidth and topology used for this can be found in Publication II.

Our evaluation shows some preliminary results about FADES, which were also crucial to spot limitations in our initial work and to open a discussion on how to address them in our future work. In Section 4.6, we discuss the limitations in our original design and elaborate on the introduced changes.

4.5.4 Discussion and Limitations

At the time of its development, FADES showed some limitations originating from practical issues with the MirageOS unikernel. Nevertheless, it had the crucial role of laying the groundwork on which we built a more advanced version of our system. Therefore, we modified the original system design also to improve its performance and capabilities, and enable the possibility to create *chains* of unikernels: a mechanism to build distributed execution flows across multiple edge devices in the network. In fact, one of the strengths of edge computing is the possibility to distribute computation across multiple devices in order to leverage the *data locality* principle (described in the previous section) and avoid to pipe raw data directly to cloud DCs. The concept will be described in the next section together with the refined version of our system.

4.6 ECCO: Edge-Cloud Chaining and Orchestration Framework

FADES showed the power of our edge computing platform backed by results emerged from our evaluation. However, the limited scope of our preliminary work prevented us from generalizing and highlight additional insights about our solution. Based on the experience gained from this work, we developed an extended and augmented version of FADES called ECCO which is a system supporting the deployment of what we call *edge-cloud pipelines*. To do so, we devised a computational model to dynamically create distributed execution chains spanning both cloud datacenters and edge devices. This concept is crucial to support applications which require data scattered across multiple locations in the network combined with multiple, intermediate processing steps. To support such applications, we posit the deployment of a networked computing, sensing and actuation infrastructure similarly to [202]. The challenge then becomes how to enable developers to write and deploy distributed applications on a potentially shared infrastructure in an efficient way. In fact, the system itself should take care of distributing the application without having the developer adjust their code to match the infrastructure characteristics. This calls also for extra system intelligence to manage services deployed by multiple providers competing for hardware resources at the edge. In this part of our work, we build on top of these requirements to upgrade FADES and deliver the following improvements:

- A distributed computational model supporting edge-cloud offloading with unikernels.
- Restructuring of FADES's initial design in order to improve its performance and streamline the implementation.
- An edge function chaining protocol to create multi-node, edge-cloud execution pipelines on-demand.
- A pass-through library for unikernels enabling access multiplexing to the edge node hardware interfaces (e.g., sensors and actuators).

In order to better describe our system and contextualize it to practical applications, we frame our discussion around a set of sample use-cases leveraging the roadside infrastructure in order to improve the services delivered to road users. For example, delivering detailed metropolitan maps coupled with citywide pollution fingerprinting to improve citizen health or helping pedestrians and cyclists select less polluted routes. However, our system is not tied to this specific application scenario and can be applied also in other contexts.

4.6.1 Model of Computation

Our model of computation is based on three core elements: the — potentially heterogeneous — inputs received by the edge infrastructure, the functions *(edge functions, EF)* manipulating them, and the outputs enabling different services. Figure 4.11 illustrates

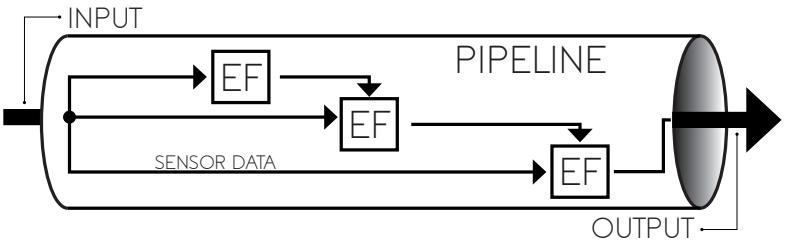
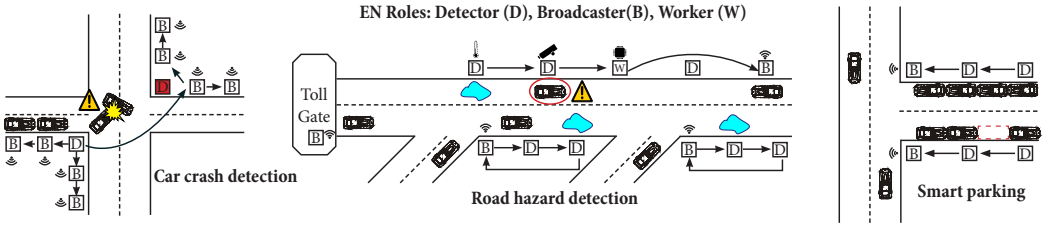


Figure 4.11: Visual representation of an ECCO pipeline.



(a). Car crashes on the road can be detected with the support of cameras. The detector node will pass the information to all nearby broadcasters to send a notification to as many vehicles as possible. In this case, a redundant detector node is deployed to counter possible node failures.

(b). Multiple broadcasters notify vehicles entering the highway of the presence of possible hazards ahead of their path. For instance, a car driving against traffic and the presence of black-ice on the road. The former requires a normal camera while the latter is more complex. Photodetectors and thermopiles, infrared cameras (thermography) or Peltier elements installed directly below the asphalt are possible options. In this situations, allocationg different tasks to each detector node is required by their different sensors.

(c). Detectors nodes (D) scan detect the presence of free parking spots directly below them. The detection can be done with cameras, IR proximity sensors or ultrasonic sensors. The acquired information is then disseminated to nearby vehicles by the broadcasting nodes (B).

Figure 4.12: ECCO example use-cases. (a) Car crash detection, (b) road hazards detection, (c) smart parking.

how these elements are connected to form an ECCO pipeline. In order to better describe our distributed model of computation, we take advantage of three practical examples based on the smart road infrastructure. Specifically, we choose (a) car crash detection, (b) road hazards detection, and (c) smart parking as shown in Figure 4.12.

Running a pipeline requires infrastructure on which to execute it, planning to select the appropriate resources on which to instantiate it and, furthermore, orchestration to move compute tasks to the suitable nodes as required by, e.g., the end-user making use of the service. We next describe the pipeline components (network, functions, nodes) in more detail, as well as their deployment and execution strategy.

4.6.1.1 Pipeline Components

In our scenario, the road users receive the pipeline output. For example, autonomous vehicles might receive the information processed by the infrastructure via long-range communication radios such as LoRaWAN [203] or LTE-V2X for Vehicle Fog Computing [204, 205].

Edge Nodes (EN). We define an EN as a device close to the end-user, such as a mobile phone, PC, or wireless access point. In other contexts, the definition might be

extended to include Radio Access Network (RAN) micro-servers [206]. In our case, we focus on the roadside units or equipment which are deployed on the road to monitor it and collect data. As they are stationary, we assume good connectivity to the cloud and to other ENs forming what we call an *edge network*.

Edge Functions (EF) are self-contained, atomic[8] functions embedding a small piece of the application logic that can be executed standalone. When chained together, they form what we call a pipeline. Each instance of EF plays a specific role and is hosted on a EN. They need to be placed based on the available data sources, the current load of the hosting device and the geographic position.

Edge-Cloud Pipeline. The pipeline describes the execution of a distributed task involving a set of ENs. Each EN listed in the pipeline takes part in execution chains and collaborates to run it. The principles and rules behind the deployment of a pipeline will be described next.

4.6.1.2 Pipeline Deployment

We envision two levels of control in the pipeline deployment and management process: (*i*) the cloud, which defines the high-level, application-driven pipeline deployment plan and (*ii*) the edge which locally makes scheduling decisions based on the parameters described in the remainder of this section. The details of a pipeline structure are defined by the cloud provider, which also monitors its execution.

We assume that each EN is reachable from the cloud and can report its status and current load in terms of active EFs and pipelines. Based on this, the service provider can plan a pipeline based on a set of parameters shown in Figure 4.14 and of which some will be described as follows.

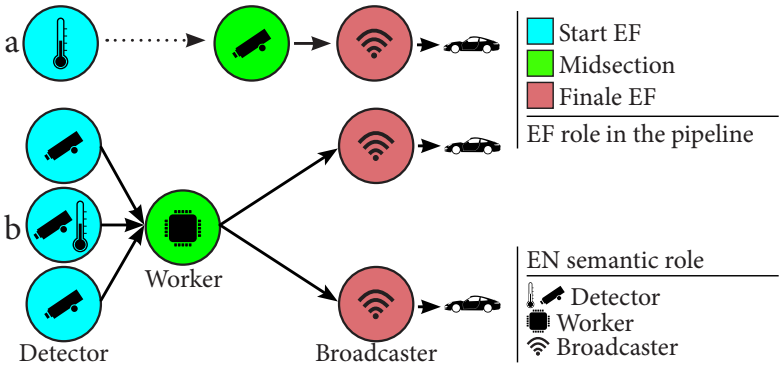


Figure 4.13: Pipelines' execution graphs based on ENs capabilities and EFs roles.

ENs have limited resources that are shared by multiple concurrent services. A scheduling procedure is required in order to make decision regarding which services should be executed at a specific time based on policies established by the service provider. Additionally, such policies are matched with the local status and capabilities of the edge

[8]They are *atomic* in the sense that, once started, they execute without external interruption **and** cannot be further subdivided without breaking their purpose as a service.

nodes which has to, eventually, multiplex multiple tasks. Parameters such as *priority* and *execution* present in the pipeline configuration decide when a pipeline should be executed. The *priority* field assumes different values based on the application criticality. For instance, car crash detection will always have higher priority than smart parking. This information allows the system to dynamically terminate low priority services when additional resources are demanded by the high priority ones.

The parameter *execution* can assume two values: on-demand and automatic. On-demand pipelines are only deployed when requested explicitly by the service provider. For example, only during a specific time window. Conversely, pipelines flagged as *automatic* are dedicated to safety critical applications which will be scheduled to run constantly. Additionally, deployed pipelines can run indefinitely or for a specific amount of time. The *deadline* parameter specifies how long a pipeline should be running on a set of nodes.

Pipelines are flexible and adapt based on the available ENs. The execution flow can be represented as directed acyclic graph (DAGs) or directed cyclic graphs with topological ordering [207]. In some cases, branching of the execution flow (as shown in Figure 4.13b can happen to support specific use-cases or address corner-cases. For instance, in Figure 4.12a the pipelines branch to disseminate the alert regarding a car crash to as many repeater nodes in close proximity. If an EN is unreachable, a substitute is found to replace it or the pipeline is adjusted to skip and remove it from the execution tree. Specifically for Figure 4.12a, this means that some vehicles will not be notified if the failure affects a broadcaster EN. Conversely, if a detector node crashes, there will be another node able to detect the car crash. Branching can be eventually used to introduce redundancy in the computation to achieve a raw form of consensus and circumvent well-known problems of result validation in unreliable distributed systems [208, 209, 210].

Branching can happen also as a function of the hardware resources available at the edge. For instance, one EN might only have cameras, another one only a proximity sensor and a broadcasting interface. This information is collected by the cloud and used to opportunely plan the pipelines structure. ENs without a broadcasting interface can only have a detector role which in turn is defined by its sensors' capabilities. By analogy, there can be ENs with both the detector and broadcaster role. In other cases, we might need an additional worker node to perform a computationally intensive task. These are ENs offering more computational power than their peers or equipped with specialized hardware (e.g., GPU). An example could be the integration of feeds of different type of sensors to detect hazard on the roads which requires additional compute power and resources as shown in Figure 4.12b.

4.6.1.3 Pipeline Execution

The pipeline configuration shown in Figure 4.14 is generated by the cloud provider once the execution plan is ready. The ENs receiving it will parse it and identify the sections they can execute in relation to the other nodes. Each pipeline is thus split into sub-pipelines, and transformed into multiple executable stages. Execution order of EFs

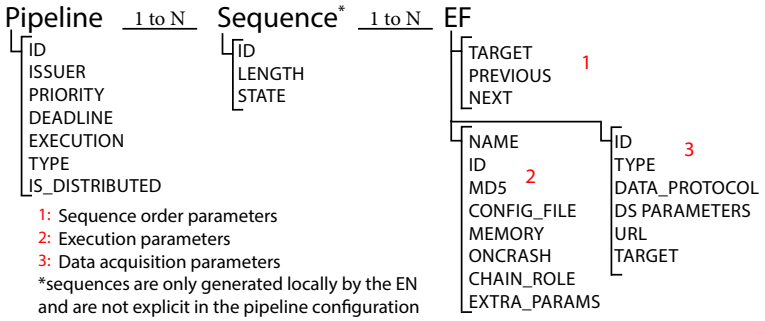


Figure 4.14: Pipeline configuration.

within an EN can be based on various parameters, e.g., priority, expected load, and deadline.

ENs scan the received pipeline configuration and each identifies the group of EFs it should execute. The classification determines the order to execute and chain EFs, and the role of each EF. An EF has one of three roles: *(i)* Start, starting a sequence; followed by *(ii)* zero or more Midsections; culminating in *(iii)* a Finale which closes the sequence. Sequence ordering parameters are used to correctly unfold execution onto the ENs. The nomenclature adopted in Figure 4.12 (detectors, broadcasters and workers) applies to the EN while the one just introduced only to the EFs and it is used internally by the system to properly order the pipeline graph. What matters for the pipeline engine is the relative execution order of the EFs and not their actual task in relation to the EN capabilities. The relationship between this two concepts is shown in Figure 4.13.

4.6.2 ECCO Design

Here, we provide an overview of ECCO shown in Figure 4.15. ECCO is designed to achieve two goals: *(i)* provide an offloading for lightweight services orchestrated by the cloud and running on constrained devices at the edge; and *(ii)* support seamless cooperation and interconnection of ENs to run distributed pipelines. ECCO retains some similarities with FADES but differs in many aspects. The changes made to ECCO aim at addressing functional limitations or improve the performance of its predecessor. Below, we describe the most important ones.

- **Merging the DRB into ECCO host modules.** One of the limitations of FADES was the overhead introduced by the communication between the Data Resource Broker (DRB) and the Data Manipulation Functions (DMFs) [9]. As the two entities communicated through a network interface, a considerable overhead was introduced in the application complexity combined with a non-marginal boot-up penalty due to the cost of loading the unikernel network library in each EF[10]. For

[9] DMF and EF are interchangeable terms. From now on, we will exclusively use the latter.

[10] In our tests and with the MirageOS version we used, this value ranged between hundreds of milliseconds and multiple seconds, depending on the underlying hardware.

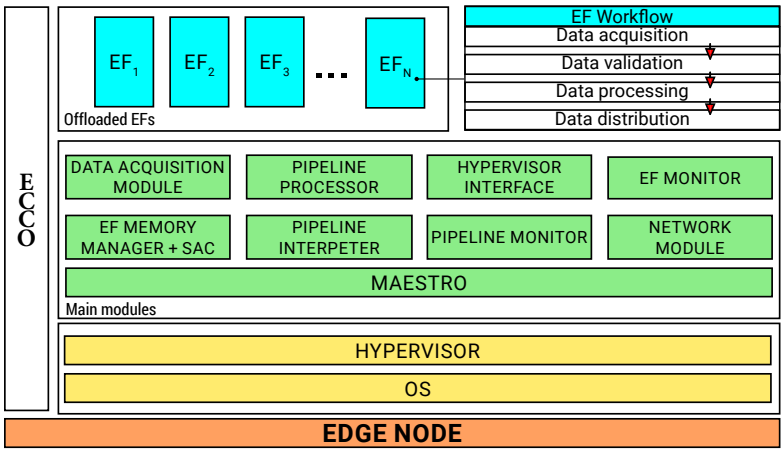


Figure 4.15: ECCO design.

the DRB, it was necessary to progressively enrich it with new libraries and connectors required to access different external data sources. This violated our principle of keeping the virtualized instances running on our framework task-specific (inspired by the libOS principle described in Section 4.1.3) by steadily increasing their complexity. Moreover, the use of the DRB proved to be an unwanted approach that would introduce a single point of failure in our system.

- **EFs Communication.** The nature of communications in a distributed system is an important aspect as increased coordination and cooperation demands more information to be exchanged at a higher communication rate. Protocols that keep this form of *synchronization overhead* to a minimum are desirable. The spectrum of communication methodologies can be delineated into three areas: the paradigm by which communication takes place, the semantic content of the information, and the protocols adopted [211]. We discuss here the first one which is the most relevant for the changes we introduced in ECCO. Generally, the paradigm by which communication takes place in distributed system is either shared global memory, message passing, or a combination of these. In ECCO — as we wanted to avoid using network interfaces to have the EFs communicate — we opted for a shared memory approach. Precisely, we used a single-layer *blackboard* [212] model where the shared memory is viewed, in fact, as a blackboard on which to write and read messages or results. To reduce the communication overhead, each EF was notified of the memory blocks to use to communicate with other EFs. By making use of watchdogs, the EF would wait until the data would be available before executing its computation. Therefore, the execution flow is event-driven or, equivalently, data-driven as the presence of new data at a specific location is a trigger starting the EF. Additional synchronization channels are also available and are only used in case of data transmission errors similarly to *expectation-driven* communication [213] (and potentially other models with non-conflicting agents such as *tacit bargaining* [214, 215]). With this approach, communication

overhead is minimized when an agent has a model of the state of other agents and, therefore, only needs to communicate when that model incorrectly reflects an agent's perceived reality [211]. This matches perfectly our communication protocol as each EF follows the same workflow, which will be described in the next bullet-point.

Regarding ECCO, depending on pipeline structure, data can be exchanged in two ways: *(i) Intra-node* communication occurs when the transfer involves two co-located EFs or an EF and the host module; and (*ii*) *Inter-node* communication occurs when the transfer takes place between two EFs on different ENs. For the former, a set of parameters is provided to determine the source and destination addresses of the shared memory page blocks used to transfer the data. For the latter, the ECCO host module takes care of transferring the data over the network to the destination EN. In fact, EFs are unable to fetch data directly from the local or remote source; they exist in a completely sealed environment. Hence, the ECCO data acquisition module must retrieve the required data specified in the pipeline configuration. In case of access to sensors, specific hardware registers can be exposed to the EF which then will be able to directly access them.

- **The EF Workflow** is composed of four phases: *data acquisition*, *data validation*, *data processing*, and *data distribution* as shown in Figure 4.15. This is similar to the workflow for a DMF in FADES, but it has been modified in order to be compatible with ECCO's distributed computing model.

  During the data acquisition phase, an EF waits for the necessary data from ECCO which identifies the correct data source and retrieves the data on behalf of the EF. In fact, ECCO exposes to EFs different end-points to access sensors or local databases identified during the bootstrapping phase. The specifics of the data retrieval phase depend on the datasource type. For instance, in case of hardware sensors, the driver code is embedded directly into the EFs, while for external sources (e.g., databases) ECCO would use host libraries to read the data and transfer it to the EF. The data validation phase checks the received data for errors, possibly requesting a re-transmission. The data processing phase is the core of the EF as it contains application logic code. By customizing this part of the EF, it is possible to execute arbitrary code in the EF, provided that the required external dependencies and libraries are available. Finally, the data distribution phase determines whether the result of the computation should be passed to ECCO or propagated to the next EF waiting in the local pipeline.

- **Sensor Access Controller (SAC).** Access to physical components such as sensors and actuators is often necessary in many use-cases associated with edge computing. In such situations, it is desirable to offer a unified interface to access such resources. Moreover, due to the presence of multiple services which might compete for the same hardware component, access multiplexing to the available physical components is required. ECCO allows EFs to directly access data stored into registers of GPIO sensors and actuators. Additionally, when multiple EFs try

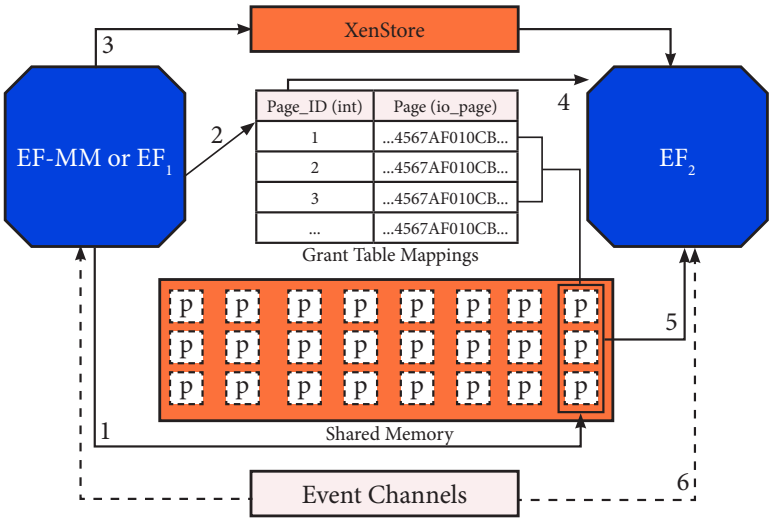
to access the same sensor, ECCO takes care of scheduling them by tracking which sensors are currently in use by the running pipelines. Another advantage of this approach is that the driver logic necessary to manage the peripheral is completely embedded into the EF. Hence, no assumptions are necessary regarding the drivers available on the host machine.

4.6.3 ECCO Implementation

In this section, we will provide implementation details for the components discussed in the previous section as they are the biggest modifications made to FADES. More details can be found in the attached research paper (Publication VIII).

- **EF Pipelining.** Before an EF can send or receive data, it needs to know its memory space *coordinates*. Regardless of the exchange between host and EF or two EFs, ECCO makes use of the communication channels offered by XenStore, a key-value storage space shared between virtual domains running on top of Xen. It allows to exchange details about the memory location and size of the data required by an EF. Access to specific key/value pairs is granted per EF to reduce malicious access by non-authorized virtualized instances. We designed a communication protocol to initialize, mediate, and terminate the transfer of data between two EFs. This protocol was built on top of Xen event channels, the basic primitives provided by Xen for event notifications. For every transfer, four communication channels are used. These are closed when no longer needed in order to avoid channel pool depletion.

- **EF Memory Manager (EF-MM)** provides a generic mechanism to share memory pages between domains. Each domain has its own grant table and ECCO allocates to each EF a set of memory pages. Then, it maps them into the grant table and receives back a list of mappings. Through the XenStore, the list is shared with the associated EF which is then free to access the content of the shared pages. It is a zero-copy procedure where only the page ownership is modified by setting specific access flags. Figure 4.16 shows the sharing data process between ECCO and an EF. The procedure is the same when transferring data between two EFs with the only difference being the use of Xen event channels to support the synchronization protocol. In order to enable this mechanism, we made extensive changes to a set of functionalities of MirageOS.



Step	Description
1	Data is mapped into the shared memory pool.
2	The EF Memory Manager or EF_1 populates the grant table with the mapping details.
3	A bundle containing details about the mapping plus other information is shared with EF_2 through the XenStore with a specific key and access policy.
4	EF_2 reads from the XenStore the page_ids and accesses the grant table.
5	EF_2 maps into its memory space the granted memory pages and accesses the content.
6	(Optional) The two EFs use the event channels to verify that the exchange was successful.

Figure 4.16: ECCO EF-MM implementation.

- **Sensor Access Controller (SAC).** It manages the general-purpose input/output (GPIO) registers to read/write the state of sensors or actuators. Sensors are accessed with GPIO operations. Once the correct GPIO address is identified, we map the respective physical page into the virtual memory space. Depending on the architecture, as shown in Figure 4.17, a two- or three- steps address translation procedure for ARM or x86 is required, respectively. After the translation phase, the page is mapped into the grant table and shared with a specific EF. Each physical register is shared inside a memory page which is similar to how the OS maps physical registers. Access granularity is down to the register bits and it is achieved using a procedure called pinmuxing [216]. This allows to map to a GPIO pin name a GPIO pin multiplexing name by using a specific pin multiplexing table. The procedure is fairly complex as it requires a combination of base address and offset value for each GPIO device connected. As there are multiple registers per GPIO device (e.g. nine for the Tegra186 chipset [217]), the offset value is used change values of registers controlling specific properties or send commands to the GPIO device (such as on/off for simple sensors).

  In the current implementation, we focus only on sensors that can be controlled by modifying the GPIO data registers. Nevertheless, we port the driver code of a

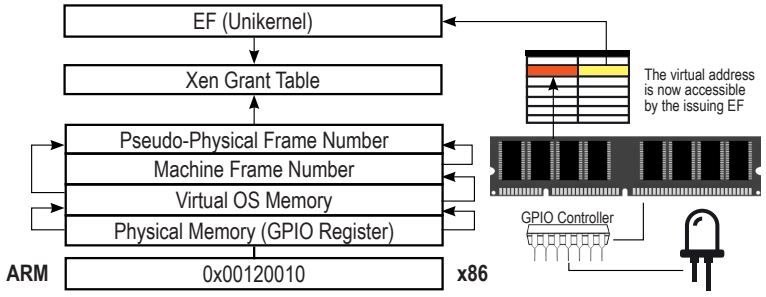


Figure 4.17: SAC memory translation procedure.

micro-controller based sensor (DHT-11) into Mini-OS and wrapped the complete driver code into an unikernel. Proper functioning of the kernel module depends on knowledge of the physical addresses exposed by the GPIO controller and the correlation with the physical GPIO pins. To each GPIO device a set of registers are assigned by the GPIO controller that can be used, for example, to control the device behavior.

In the next Section, we will provide an evaluation of ECCO and discuss in details the results emerged from it.

4.6.4 Evaluation

We evaluated ECCO from different angles but here we will focus on providing insights primarily regarding the pipelining process and the functionality connected to it. Different devices were used to understand the performance gap between the edge and cloud. In our tests, the edge devices were comparable to micro-servers rather than base stations. The nodes used in our tests were an Intel NUC (NUC), Dell Optiplex (OPX) and two high-end Dell PowerEdge 730 servers (SRV1, SRV2), all connected to the same LAN network and for which detailed specs are provided in Table 4.1. The results obtained from the experiments are averaged over 100 repetitions.

Table 4.1: Devices specifications.

Device	CPU	RAM	Ocaml	Xen	OS
Dell PowerEdge R530 (SRV1, SRV2)	Intel Xeon E5-2640 2.60GHz — 32 Cores	128 GB	4.04.2	4.6.0	Ubuntu 14.04 Kernel 3.19.0
Intel NUC (NUC)	Intel i5-6260U 1.80GHz — 4 Cores	16 GB	4.04.2	4.6.6	Ubuntu 14.04 Kernel 4.4.0
Dell Optiplex 7050 (OPX)	Intel i5-7500T 2.70GHz — 4 Cores	8 GB	4.04.2	4.6.6	Ubuntu 14.04 Kernel 4.4.0

ECCO is profiled under different loads, with different devices, and varying pipeline topologies, as shown in Figure 4.18. For each case, we use the same abstraction model

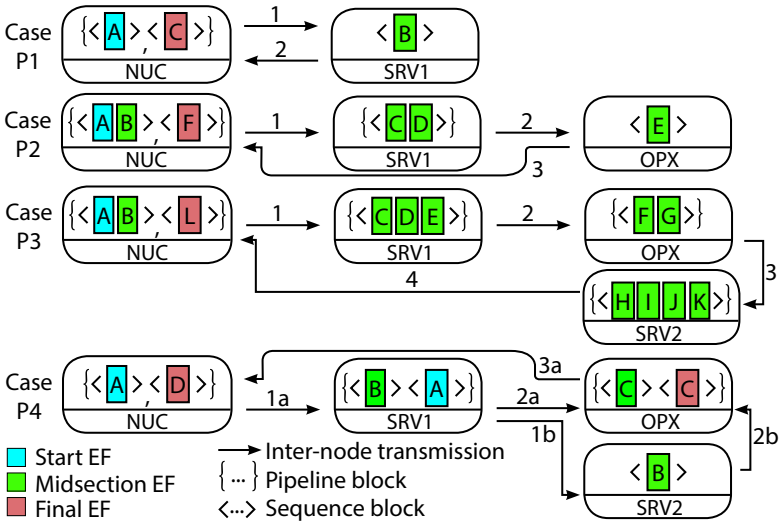


Figure 4.18: ECCO pipeline topologies.

introduced in the previous section: the starter node is always both a detector and broadcaster, the remaining ones are only detectors. Case P1 represents a pipeline involving only two nodes, and the results reflect the processing load of the edge computing nodes; P2 and P3 are pipelines with increasing complexity having more ENs and EFs. This allows us to profile our system under different configurations. Each pipeline begins and ends with Start and Final EFs which are identified by the blue and brown blocks, respectively. The green blocks represent Midsection EFs deployed in the middle of the execution chain. A pipeline block is delimited by braces and contains sequences of blocks delimited by angle brackets. Case P4 has two pipelines delimited by curly brace blocks. Each node executes only the blocks assigned to it. These will be executed in a specific order (represented here with a character in each block). For instance, in Case P1 we will execute first the EF with tag **A** on the NUC. Then, the output of the computation will be sent to the EF with tag **B** running on SRV1. Finally, the output will be transferred to the EF with tag **C** on the NUC. After this step, the pipeline is completed. When there are multiple EF having the same tag, it means that they start in parallel (Case P4). For our benchmarks, we measure the pipeline completion time, shown in Figure 4.19 using an EF running simple image processing operations. The EFs exchange a complete post-processed image instead of a single value. This configuration is typical in situations where the preprocessed data has to be aggregated downstream by a worker node. Details about the EF performance can be found in the respective paper [8].

Figure 4.19 displays the Empirical distribution function (ECDF) of the edge-cloud pipeline execution time for the cases presented in Figure 4.18. The pipeline execution time includes the computational and memory overhead time in addition to the network inter-node transfer time and the unikernel boot-up time. The pipeline execution time grows roughly linearly with the number of EFs. Even though they are not directly comparable, the first three topologies (P1 to P3) differ in the number of EFs and nodes, while

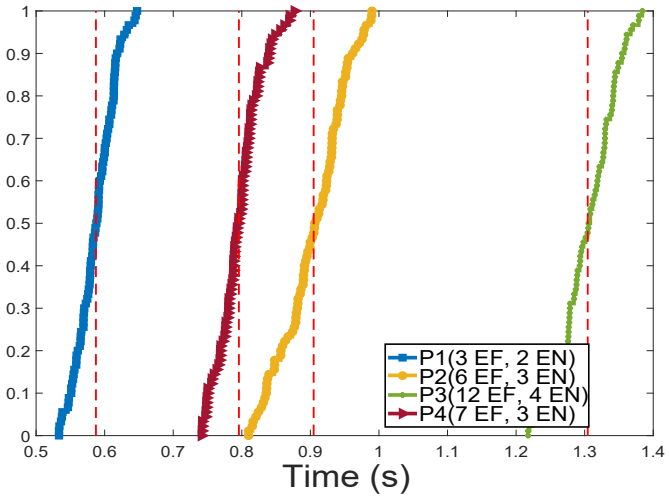


Figure 4.19: Pipeline execution time — ECDF.

the pipeline workflow remains roughly unchanged. We note that hardware differences do not play a major role since performances is similar. In each case, we double the number of EFs and add one EN. By adding more nodes we reduce the overall completion time by leveraging the parallel compute capabilities of many ENs. Additionally, this reduces the slope of the linear growth. This is important in long pipelines, where the EF to EN ratio has a noticeable effect. In other words, balancing the amount of EFs allocated to each EN have a non-trivial effect on performance. Stacking multiple EFs on a a single EN is not advised, as we are not taking advantage of the system carnality but rather overloading a single machine. We will come back to this important aspect in a different context later on in Section 4.8.1.

4.6.5 Discussion

ECCO brought many improvements over the first version of our framework including the possibility to distribute computation across multiple edge nodes leveraging the data locality principle. In such a distributed environment, one important challenge to address is how to migrate a stateful EF while preserving their state. In fact, edge and IoT networks may be unreliable [218] and composed of resource-constrained devices that are more prone to failures. Ideally, the edge infrastructure should be able to self-adapt in case of malfunctions and quickly move the computation to a stable node in order to maintain high service responsiveness and avoid data loss. Therefore, service migration and recovery play a crucial role in strengthening the reliability of an edge-cloud infrastructure especially for stateful services. This is the problem that we will address in the next section by presenting a framework supporting stateful migration of unikernels through state checkpointing. This is our last contribution dedicated to the unikernel ecosystem.

4.7 MirageManager: Enabling Stateful Migration for Unikernels

As discussed before, unikernels are a new form of Lightweight Virtualization (LV) that can be successfully leveraged for service provisioning at the edge. However, the unikernel ecosystem is still in its infancy and lacks critical functionalities found in other virtualization technologies. In particular, stateful migration is a highly desirable feature for mobile edge services in distributed environments which is not yet supported by unikernels. In the work presented in this section, we introduce MirageManager: a ready-to-deploy unikernel migration system enabling lossless migration supported by a function-level, application logic checkpointing library.

4.7.1 System Design

We added the required migration logic directly at the application layer instead of making any changes at the kernel level (as MiniOS [219] does) or in the hypervisor. This is a practice also followed in past work for VM-independent migration of stateful applications or to capture the application state before migrating it [220, 221].

We designed a set of functionality in the shape of a library allowing the unikernel to keep track of its own internal state. When the unikernel needs to suspend, it serializes its state so it can be transferred to the migration target, which will process the state before proceeding with the execution flow. Aside from state tracking, we require an additional component to support the migration process which we call MirageManager. It is a web service exposing an interface to commission and manage unikernels on any registered host and transfer the unikernel state using a repository.

MirageManager is the core of our system: it manages the life-cycle (e.g., creation, migration, destruction) of unikernels deployed on multiple hosts and it provides a repository for writing and retrieving the unikernel state before and after a migration. The set of information necessary to describe an unikernel is generated before the hypervisor creates the guest domain and will exist even after its destruction, regardless of whether it is the result of a migration procedure or a regular shutdown. During the guest lifetime, the representation will change to reflect changes in its state. Once the service embedded in a migrated unikernel has terminated its life-cycle, also its previously checkpointed state stored in the MirageManager repository is removed.

Figure 4.20 gives an overview of the migration procedure. When an unikernel is created, MirageManager will create a guest domain of the corresponding image on the target host. Afterward, to confirm a successful boot, the unikernel queries MirageManager and start a lookup procedure for a previous state associated to it. This procedure is required so that, even if no state is retrieved, MirageManager is aware of the current state of the unikernel (specifically, started) and change it to connected. Therefore, every newly booted unikernel will at least once interact with the MirageManager to notify it of its existence on a specific machine. At the moment of a migration, MirageManager issues a suspend command to the unikernel. Hence, the latter transfers its state to the repository and thereby confirm that the suspension was successful. On the target machine,

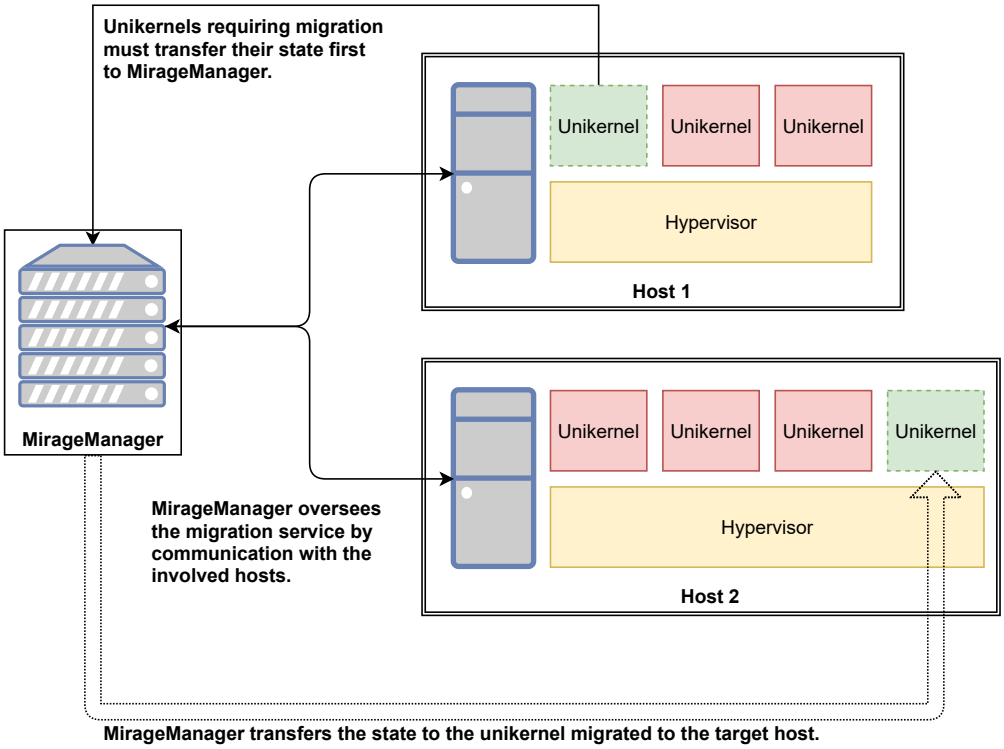


Figure 4.20: Unikernel migration with MirageManager.

a unikernel with the same image is created, a previous state is retrieved from MirageManager and the just created guest unikernel uses it to update itself before resuming its workflow exactly from where it was interrupted before the migration. This process can be repeated indefinitely until the unikernel is permanently stopped, completes its intended task, or exits due to an error/fault. Potentially, the unikernel could invoke by itself the migration procedure without the need for an external trigger.

4.7.2 State Checkpointing

In order to manage the lifecycle of a unikernel, MirageManager requires a complete representation of its execution state. Therefore, we developed a module able to store and serialize the unikernel application logic state so that execution can be resumed from it. We call this procedure checkpointing and it is described as follows. For the purpose of creating checkpoints, we implement a library for MirageOS that defines a central state object representing the state of the unikernel. Additionally, we defined a programming model that allows to express the application logic routines in a serializable format, so that the execution state can be written to the store and transferred to the repository.

This store is called *Unimem* and it is implemented as a key-value store using strings as keys and a polymorphic data-type for the values.

To enable the application to write its execution state to Unimem, we translate the unikernel into a series of atomic procedures, each constituting a step. Hence, we define the application workflow as a directed graph with labeled edges where each node is a computational step. Every step is identified by a unique string identifier (ID) so that the currently active step can be dumped into the store just by using its ID. Edges are guarded by expressions using the variables present in the store that determine how the control flow is directed from one step to the next.

In case multiple edges originate from the same computational step (e.g., execution logic branching), the program decides which one to follow by evaluating what we call *transition guards*; these are conditional equations evaluated on variables stored in Unimem. Therefore, the control flow can be expressed as an adjacency matrix where each entry a_{ij} describes the transition from step *i-th* to *j-th* and its value acts as a guard for the transition. The truth condition of the transition guard is obtained from evaluating specific functions on a set of variables in the store. The first condition evaluating to true in a row of the adjacency matrix determines the next transition in the execution flow. One limitation is that guard functions must be mutually exclusive to avoid ambiguities in the process of selecting which transition to take at any given moment. If no guard condition evaluates to true, the application logic is considered to be completed and the unikernel terminates. This is very similar to state machine models [222] which in our case match well the compile-time defined behavior of unikernels.

When an unikernel is requested to suspend or migrate, the identifier of the currently executed function is written to Unimem. As every variable used to evaluate the transition guards is stored as well, the content of Unimem fully describes the application state. In fact, the current position in the graph as well as the next transition to be traversed can be inferred from the stores content only. To protect against state corruption and potential information loss, all variables belonging to an execution scope spanning multiple computational steps must be stored. This is facilitated by not using return values or parameters for the steps, but rather writing from and reading to the store. As computational steps are atomic, the information contained in Unimem is sufficient to recreate the application state after a migration. Finally, there is no specific structure imposed on the content of Unimem by MirageManager as long as the state is serializable. Therefore, also other state information, such as the state of an object-oriented application, could theoretically be stored.

4.7.3 Implementation and Migration Workflow

MirageManager is implemented as a distributed system consisting of an application server developed with Express [223] and written in JavaScript. It exposes a REST API to issue migration commands and for the unikernel to transfer its state to the central repository. Additionally, each host wishing to use MirageManager needs to run a controller so that the central application server can communicate with the hypervisor on that machine. In Figure 4.21, the ServerNode hosts the application server while the

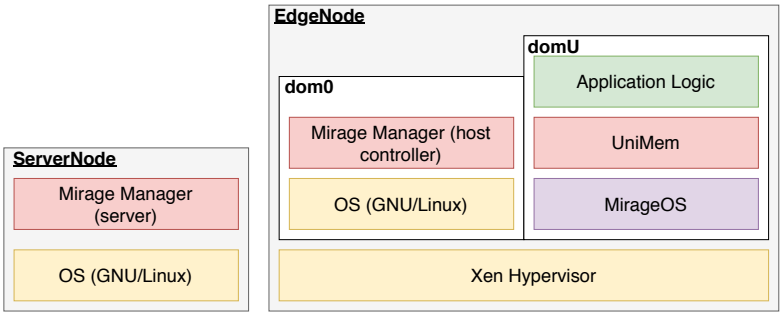


Figure 4.21: MirageManager system components.

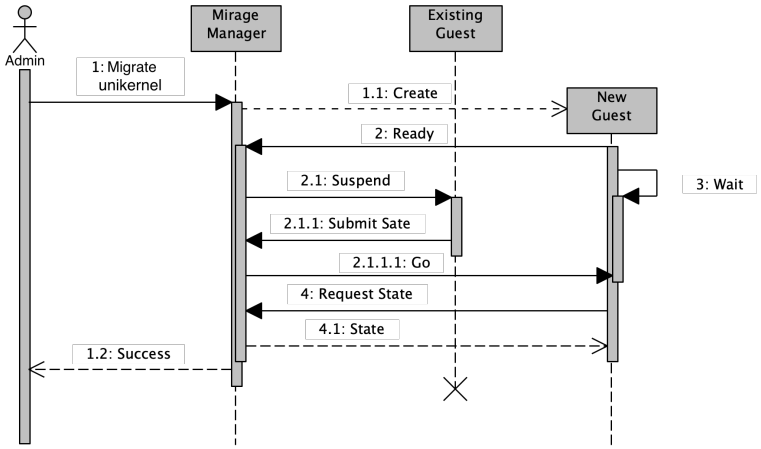


Figure 4.22: MirageManager migration workflow.

EdgeNode is as device using the migration functionalities. In our implementation, we used Xen as hypervisor [159] but other options are possible.

The migration procedure follows a set of steps shown in Figure 4.22 for which additional details can be found in Publication III.

4.7.4 Evaluation

To evaluate MirageManager, we selected as baseline Podman which is an engine for running Open Container Initiative (OCI) containers with support for Checkpoint/Restore In Userspace (CRIU)-based migration. We embedded an application with the same functionality as our MirageOS unikernel inside an OCI container and then performed the migration tests. The application we used to test our system was a simple incremental counter which was enough to prove that the stateful migration mechanism works. Additional details about the experiment setup and collected metrics can be found in Publication III. Here, we focus on service migration at scale which is an important factor in distributed systems such as an edge-cloud infrastructure. Therefore, we evaluate

how migration with MirageManager fares in comparison with Podman when both tools perform multiple migrations simultaneously and measured the overall migration time.

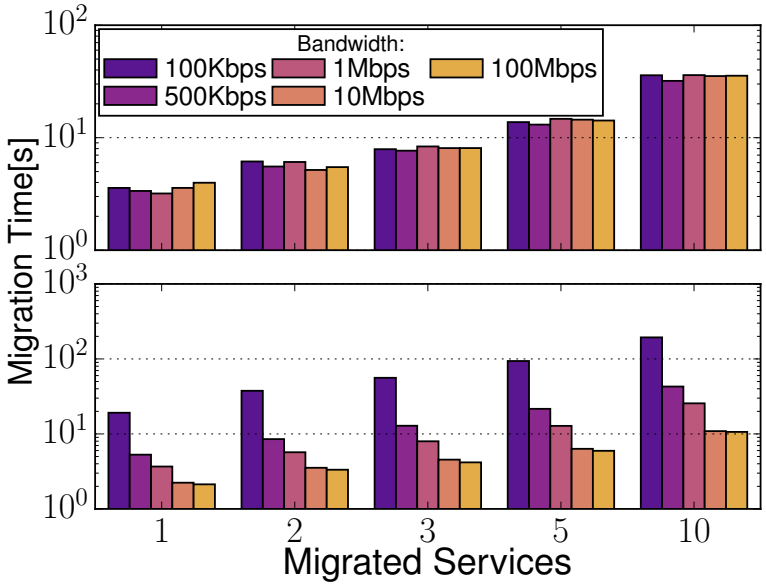


Figure 4.23: Scalability analysis — MirageManager (top) vs. Podman (bottom).

Figure 4.23 shows the overall time required to migrate multiple services in parallel. The top part of the figure shows the results for MirageManager while the bottom part shows those for Podman. For both, we measured the overall migration time with five different network bandwidth settings. Edge networks may suffer from limited network bandwidth which may severely impact migration operations when the migrated service state has a considerable size. This stresses the need not only to follow best-practices of service decomposition but also to reduce the state size as much as possible. For both, unikernels can be the answer.

We can gain multiple insights from Figure 4.23. First, MirageManager migration time is seemingly unaffected by the network bandwidth and it grows quasi-linearly with the number of services migrated in parallel. The same cannot be said for Podman, which is definitely suffering in low bandwidth conditions as it needs to transfer the complete memory dump as part of its migration technique. This tendency is exacerbated with the network bandwidth capped at 100 and 500 Kbps. In this case, MirageManager is up to **∼6x** times faster than Podman. On the other hand, Podman outperforms MirageManager as the available bandwidth increases. It is worth noticing that, as the size of the migrated service status gets closer to the Podman dump size, the migration time becomes the same for both approaches. However, unikernels usually embed simple applications which seldom require massive amount of synchronization data. Hence, we can conclude that the effectiveness of our approach also depends on the specific application and the size of its migrated state.

Finally, while migration with Podman is transparent to the migrated application, MirageManager requires changes to the application logic in order to work correctly. Based on this, we state that MirageManager generally outperforms in downtime and data transfer volume cold migration with containers while offering competitive performance in terms of overall migration time.

4.7.5 Discussion

While MirageManager outperforms well-known migration tools for other virtualization techniques such as containers, it is still open to many improvements. One of its biggest limitations is that developers that want to use our framework need to translate their existing software stack (or parts of it) into the library OS specific programming language (in this case, OCaml). Our system could be extended and ported to work with other unikernels [127, 180, 182, 224], which would bring more freedom in terms of usable programming languages. However, this would cost additional integration effort due to the different programming models. Another limitation is the additional overhead introduced by the programming style demanded by our state checkpointing library. While these restrictions can rule out using MirageManager in some cases, our framework remains the first system enabling the migration of unikernels. Moreover, our design allows to easily extend the implementation to accommodate diverse hypervisors and library operating systems.

4.8 Edge-Cloud Resource Provisioning

The three pieces of work described in the previous sections focus on system design and implementation challenges that are at the core of our research in the direction of leveraging LV techniques in the context of an edge-cloud infrastructure. In this section, we shift our focus towards scaling challenges while preserving the research context. Specifically, we focus on resource provisioning and task allocation challenges which are yet applicable to the LV framework we discussed beforehand. Additionally, we take a *user-in-the-loop* perspective to better assess the impact of our solution in terms of quality of service improvement. We conclude this Chapter by discussing our edge-cloud, resource provisioning framework called *Nimbus*.

4.8.1 Nimbus: An Edge-Cloud Allocation Algorithm for Task Offloading

Unlike other driver applications for edge computing discussed in the previous sections, real-time multimedia applications impose much stricter constraints on offloading computations to edge devices. Considering the complexities levied by deep learning-based real-time applications, it is challenging to exploit a nearby edge infrastructure in a scalable manner. However, supported by the interplay between edge and cloud, it is possible to extend cloud computing outside of datacenters, and enhance its services by leveraging an infrastructure closer to the end-users.

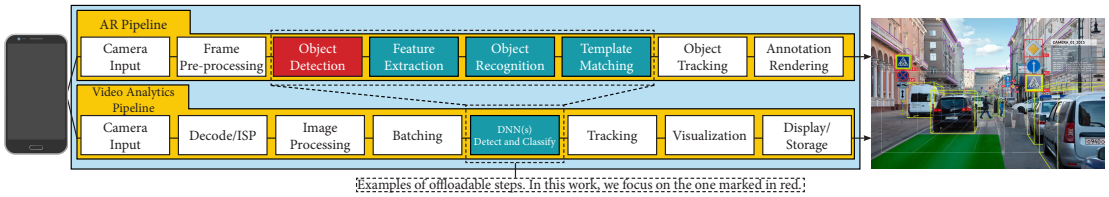


Figure 4.24: Mobile applications requiring deep learning steps.

In this context, one important challenge is how to select the best offloading candidate in order to maximize user satisfaction, as allocating multiple users to an already overloaded edge node can negatively impact the application performance. Hence, the algorithm described in Nimbus is designed as an optimal task placement strategy for real-time applications (in particular image processing), such as the ones shown in Figure 4.24. Nimbus goal is to minimize the overall mobile-to-edge latency (or maximize FPS) while avoiding the increase in battery consumption. Additionally, our algorithm aims at distributing the computational load across the edge-cloud infrastructure avoiding to saturate the resources of the edge devices. In fact, Nimbus's offloading policy ensures a balanced load distribution across the edge nodes participating in the infrastructure similarly to load balancing practices used for web servers [225].

4.8.1.1 System Overview

We consider a multi-tier, edge-cloud infrastructure similar to the model discussed in Chapter 2. Figure 4.25 shows the *entities* in our system – mobile devices (MD) and edge nodes (EN) interacting over the network. The former interact with the infrastructure as users of AR applications and the latter are responsible for handling tasks offloaded by the MDs and constitute the core of our infrastructure. We assume a hierarchical edge architecture where compute and caching capabilities of ENs increase with increasing distance from the MD.

We logically divide the network into three layers – each one offering different capabilities and, as we approach the core of the infrastructure, latency and computational capacity of resources' increase.

Tier One Edge Nodes (T1-EN). The outer-most layer (denoted by blue circles in Figure 4.25) is a set of augmented access points (AP) and routers with minimal compute capabilities. We assume these APs to be either equipped with (or directly connected to) an embedded device with low-end GPUs, e.g. Nvidia Jetson Nano or Intel NCS. Resources in this layer act as entry points to the network, offering limited computation in addition to standard routing and connectivity functionality.

Tier Two Edge Nodes (T2-EN). Shown as pink squares, they form the second layer of our multi-tier edge cloud infrastructure. Logically, these devices can be seen as backbone routers co-located close to T1-ENs. T2-ENs have at their disposal increased computational power and network bandwidth that allows them to serve multiple users

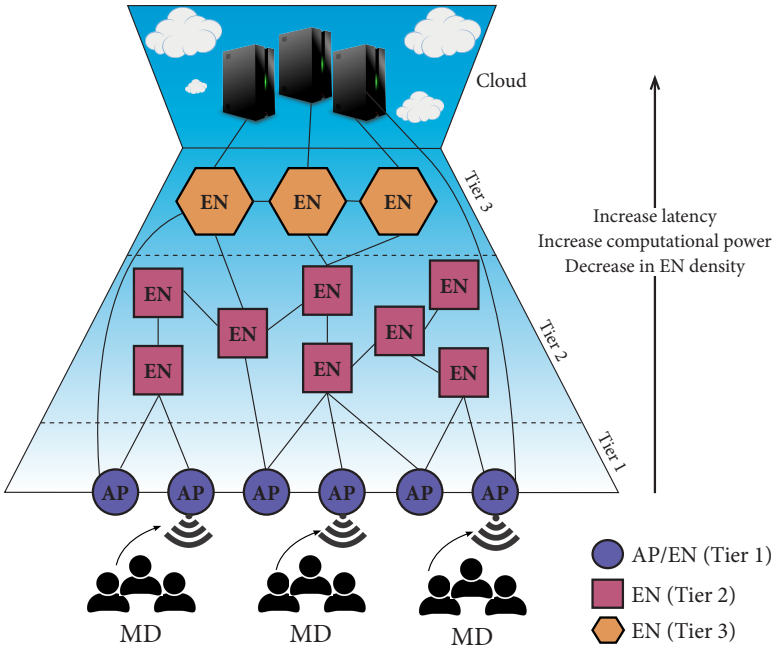


Figure 4.25: Nimbus Infrastructure.

in parallel. T2-EN resources are a mid-range micro-server equipped with a dedicated GPU.

Tier Three Edge Nodes (T3-EN). At the core of our architecture we find T3-EN (shown as orange hexagons) which are powerful servers equipped with multiple GPUs, offering the most significant computational power of all layers. The capabilities of T3-EN are similar to traditional cloud datacenters, both in terms of the number of users that can be served in parallel and available network bandwidth. However, due to their proximity to the network core, the network latency incurred to access the resources in this layer is the highest compared to the rest of the edge infrastructure.

4.8.1.2 Nimbus Algorithm

We consider a system where a controller estimates the feasibility of offloading a task proposed by a mobile device to the edge infrastructure. This controller can be either centralized or decentralised (which entails the presence of multiple controllers). Figure 4.26 shows a high-level, concise workflow representing the interaction of a MD with the Nimbus controller. It explains how an MD can offload the execution of a task to the edge-cloud infrastructure. The latter is composed of **N** interconnected and heterogeneous ENs, which, based on their computing capacity, can serve several concurrent tasks. An MD can offload its task via a T1-EN, which acts as gateway to the infrastructure.

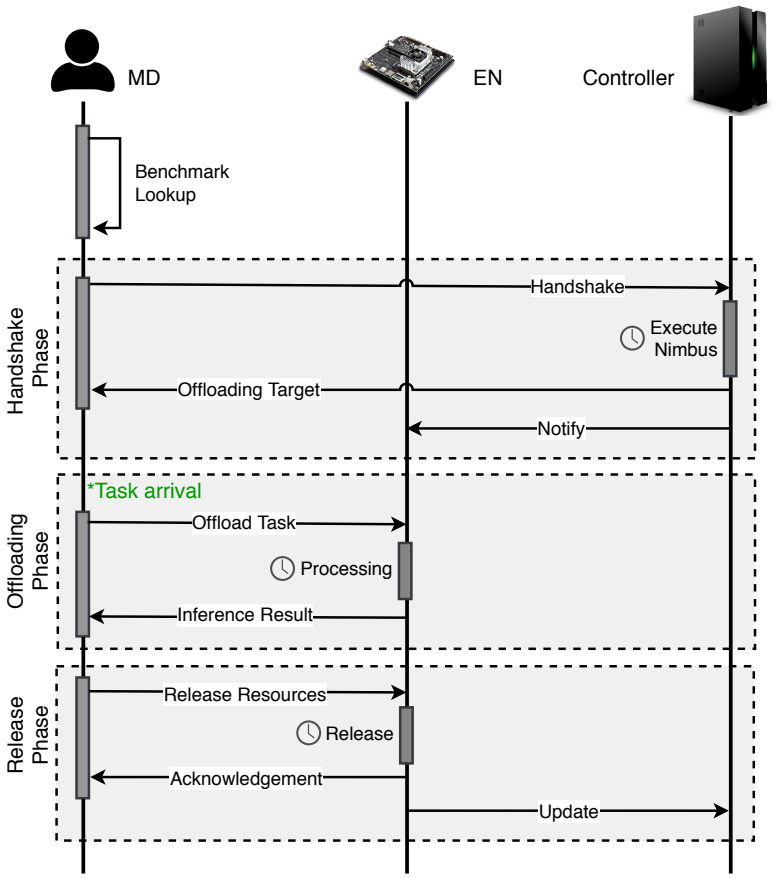


Figure 4.26: Nimbus Workflow.

The controller is responsible for making sure that the offloaded tasks are assigned to ENs with two goals in mind: minimize latency and reduce battery consumption for the MD.
Assuming that the i-th task is executed by j-th EN, the task latency and battery consumption incurred by the device can be formalized as (see Table 4.2 for a summary of the notation):

$$L_{ij} = (Lt_{ij} + Le_{ij}) = \left[(\frac{d_i}{BW_{ij}}) + RTT_{ij} \right] + (TET_j + q_j) \tag{4.1}$$

$$B_{ij} = Bt_{ij} = Lt_{ij} \times w \tag{4.2}$$

To solve for both latency and battery consumption, we make use of specific techniques to adjust the problem space and reshape a multi-obecjtive optimization problem into a single-objective function. Mathematically, let x_{ij} denote the case when the j-th EN serves the i-th device. By using the ϵ-constrained method which helps in solving multiple-criteria decision making problems [226] by limiting the solution search space, we can express the latency minimization problem as follows:

Table 4.2: List of parameters used by the algorithm.

Term	Description	Unit
d_i	Amount of data transferred by the *i-th* MD	KB
BW_{ij}	Bandwidth between *i-th* MD i and *j-th* EN	Mbps
TET_j	Inference time on the *j-th* EN	ms
TEC_i	Local energy execution cost for *i-th* MD	mJ
q_j	Queuing time at *j-th* EN	ms
w	Transmission module power	mJ/ms
ϵ_t	Latency threshold	ms
ϵ_b	Energy budget	J/s
RTT_m	RTT matrix	ms
κ, α, β	Additional coefficients	—

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} x_{ij} L_{ij}(p) \tag{4.3}$$

$$\text{subject to} \sum_{i=1}^{N} x_{ij} = 1, \forall j \in M, \tag{4.4}$$

$$L_{ij} \leq \epsilon_t, \forall j \in M, \tag{4.5}$$

$$B_{ij} \leq \epsilon_b \equiv TEC_i, \forall j \in M, \tag{4.6}$$

$$x_{ij} \in \{0, 1\}, \forall i \in N, \forall j \in M \tag{4.7}$$

where N and M are the set of mobile devices and EN, respectively, and with $\mathbf{p} = \langle\, d_i, BW_{ij}, TET_j, RTT_{ij}, q_j \,\rangle$ vector containing part of the parameters shown in Table 4.2. Equation 4.3 is our objective function. Equation 4.4 and 4.7 limit each MD to offload its task to as single EN, at most. Equation 4.5 and 4.6 are formalization of the latency and energy consumption constraints limiting the feasible solution space.

Then, we trace back our optimization problem to a convex form that we solved using a meta-heuristic. Algorithm 1 describes Nimbus's task offloading approach. The algorithm is divided into three phases. The *Warmup* phase identifies a list of ENs that are accessible from the AP the MD is connected to and are the best candidates to offload computation. In the *Core* phase, the algorithm calculates the latency and battery cost for offloading to each of EN. Afterwards, it selects the best EN based on the balance-ensuring allocator. In the *Fallback* phase, if the algorithm failed to find a suitable EN for offloading the task, it looks for a cloud server that best satisfies the latency and energy consumption constraints of the task.

Algorithm 1: Nimbus allocation algorithm.

Input : Refer to Table 4.2.
Output: Best offloading target for the *i-th* MD.

// Warmup
$\vec{EN}_r \leftarrow \text{FilterAndMinimize}(AP, RTT_m, \epsilon_t)$
$\vec{EN} \leftarrow \text{LookAheadLoad}(\vec{EN}_r, \kappa)$
// Core
for EN_j *in* $\vec{EN}$ **do**
  $L_{ij} = (Lt_{ij} + Le_{ij}) = \left[(\frac{d_i}{BW_{ij}}) + RTT_{ij}\right] + (TET_j + q_j)$
  $B_{ij} = Bt_{ij} = (Lt_{ij} \times w)$
  if $L_{ij} \geq \epsilon_t$ ***or*** $B_{ij} \geq \epsilon_b$ **then**
    $Drop(EN_j, \vec{EN})$
  end
end
if $\vec{EN} \neq \emptyset$ **then**
  for EN_j *in* $\vec{EN}$ **do**
    return $\arg\min[\alpha * \frac{L_{ij}}{\epsilon_t} + \beta * \frac{load_j}{maxload_j}]$
  end
else
  // Fallback
  $cloud \leftarrow \text{FindClosest}(\epsilon_t)$
  if $L_{cloud} \leq \epsilon_t$ ***and*** $B_{cloud} \leq \epsilon_b$ **then**
    return *cloud*
  end
end
return $\emptyset$

4.8.1.3 Evaluation and Results

We explored and analyzed multiple facets of our algorithm, namely *network latency*, *inference and queuing time*, and *specifications of MD and T1-EN*. In order to test Nimbus in realistic conditions, we conducted several experiments and measurements to collect data concerning multiple variables used in our algorithm. In particular, the most interesting ones are inference and queuing time on different GPUs as a function of the number of requests. This is necessary to understand the performance drop as multiple MDs are allocated to the same edge node. Details about these measurements can be found in the attached research paper (Publication IV). In this section, we report only a subset of the results gathered from our experiments. In particular, the improvements in terms of QoE for the end-users and trade-off of running Nimbus in a centralized vs. decentralized fashion.

Scalability & Performance. To assess the QoE improvements for the end-users, we analyze the effective task latency and energy benefits of Nimbus for processing tasks offloaded by MDs. We select four combinations of edge infrastructure and MDs. For the former, we selected a set of configurations covering scenarios with an increasing number of ENs, at different tiers. For the latter, we used a dataset providing information regarding the number of users connected to a WLAN network in a specific time interval[11]. Based on that, we grouped the users using four concentrations as shown in the x-axis of Figure 4.27. More details about the dataset can be found in the attached research paper (Publication IV).

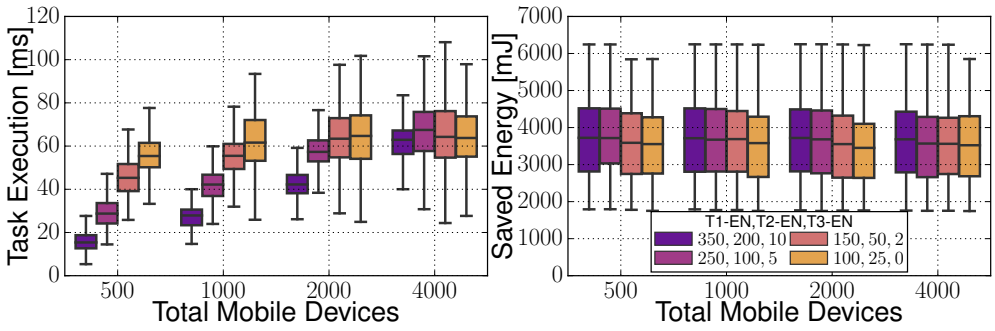


Figure 4.27: Task latency and energy saving in various setups.

Figure 4.27 is divided in two parts. The left panel shows the task execution time (in milliseconds) as a function of the MDs in the network and different configurations of ENs. The right panel shows the overall amount of energy saved by the MDs when offloading the computation (per 1 second, or 15 frames), in milliJoule. The plotted values are obtained after combining the results from 100 simulation iterations. The details of the EN configurations are indicated in the inset plot illustrating, for each color, the number of deployed T1,T2, and T3 edge nodes, respectively. By looking at the task execution time panel, we notice that even in the worst case, the expected task latency achieved by Nimbus is $\sim$**2**$\times$ lower than running it locally on the fastest MD in our dataset. From a performance standpoint, this offloading strategy can exceptionally boost deep learning based applications and increase the quality of experience for its end-users. As the number of MDs increases, the performance proportionally decreases. With more congestion and tasks offloaded, the delivered performance drops, as multiple MDs crowd the same EN and influence each other's execution time by increasing the overall queuing time. This saturation behavior is mirrored by the MDs allocation ratio.

Figure 4.28 shows the percentage of mobile devices served by the edge infrastructure, for four different configurations of ENs for which additional details can be found in Publication IV. As the number of users increases, the edge resources tend to deplete more quickly, forcing most of the mobile devices to run their computation locally or utilize the cloud. Task offloading also allows MDs to save energy (right panel of Figure 4.27),

[11]The time interval was roughly 12 months.

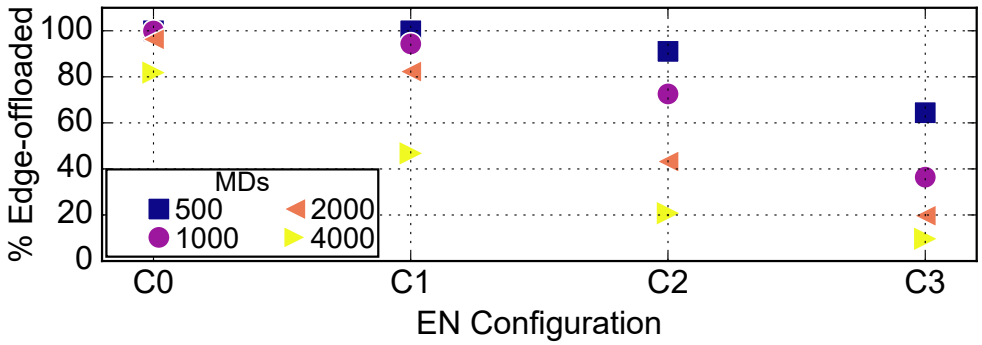


Figure 4.28: Fraction of task offloaded to the edge.

reducing the power consumption in all cases. Our results show a nontrivial margin of gain in offloading using Wi-Fi to the edge infrastructure. Even considering the most power-hungry smartphone in our dataset and the average energy saving in the worst-case, Nimbus still consumes ∼**77%** less battery.

Nimbus Variants (Centralized, Decentralized and Hybrid). We developed three versions of Nimbus. The original version was single-threaded (ST), meaning that the decision process was handled by a single controller node with complete knowledge of the edge infrastructure. From a practical viewpoint, this approach offers limited scalability, especially when both the size of the edge infrastructure and density of participating MDs increase. In this case, the convergence time of the single-threaded variant becomes prohibitive. Hence, we developed a multi-threaded (MT) variant of Nimbus, termed *MT Nimbus*, which is deployable in a distributed fashion. We applied a partitioning procedure to the edge-cloud infrastructure in order to distribute our algorithm to many solver units. Transforming an algorithm from centralized to distributed has additional cost such as synchronizing different entities increases communication overheads. Here, our goal is to demonstrate the possibility of transforming our algorithm into a distributed form and characterize its performance. While the principal benefit for our distributed algorithm is reduced time to allocated all the tasks from the MDs, we sacrifice in *quality* of the solution as the algorithm is now less capable of fully exploiting the available edge-cloud infrastructure resources. In fact, each single solver will only *see* a slice of the full network infrastructure and therefore less offloading candidates for the MDs tasks. This happens because we logically split the ensemble of ENs into non-overlapping subsets which are then associated one to each solver. The consequences of this choice are explored discussed as follows.

Figure 4.29 shows the convergence time and task execution latency with an increasing number of threads. It can be observed that the more we divide the network, the fewer MDs are offloaded because each slice becomes *smaller*, thus reducing the degrees of exploration for the algorithm. However, the convergence time per-thread reduces by up to ∼**15**× when Nimbus uses four threads instead of one.

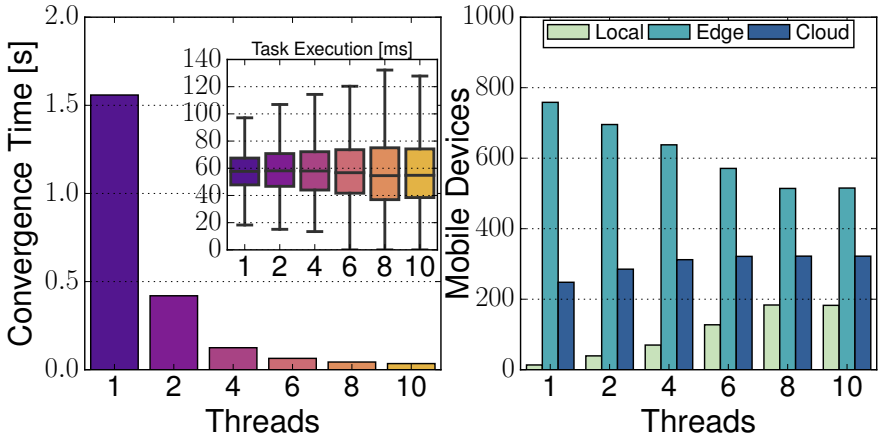


Figure 4.29: Performance of MT-Nimbus.

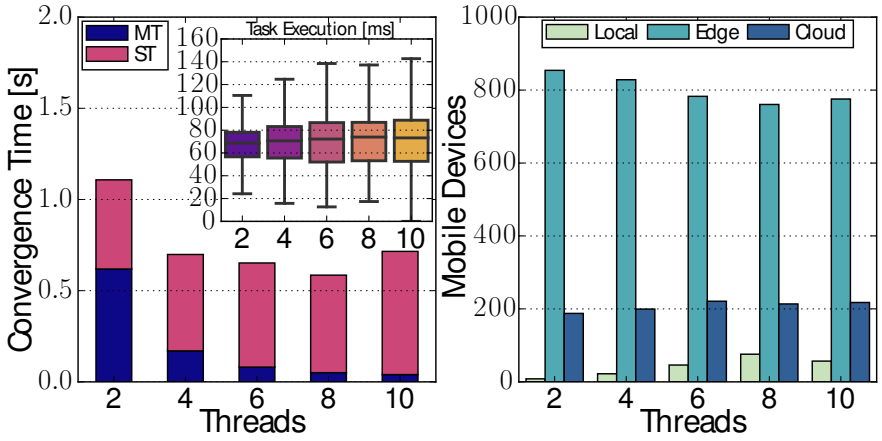


Figure 4.30: Performance of 2PMT-Nimbus (Multi-threaded (MT), Single-threaded (ST)).

To mitigate the inefficient use of the edge-cloud infrastructure, we developed a *two-stage* solver version of Nimbus. In this variant, all the MDs not offloaded in the first distributed stage are scheduled for a second allocation pass. This final variant is called *2PMT Nimbus* and the results obtained are shown in Figure 4.30.

While there is an additional cost in terms of convergence time due to the presence of a final aggregation step, the number of non-offloaded MDs reduces drastically, especially with an increasing number of threads. With only two threads, 2PMT-Nimbus achieves similar MD allocation ratios as the single-threaded version while almost halving the convergence time. With eight threads, 2PMT-Nimbus converges almost 3× faster than two-threads and offloads the majority of the users.

Note the anomaly in convergence time trend of 2PMT-Nimbus – where the convergence time increases despite an increased degree of parallelism. We explain the exception as follows. By assigning more threads, the generated network slices become shallower and

fewer EN candidates are available to allocate MDs. The fewer users are allocated, the more effort is required by the centralized solver to complete the final reallocation step. This means that the law of diminishing returns applies to the threads parallelism. In fact, with ten threads, the multi-threaded convergence time decreases, but the single-threaded increases. However, the overall performance in terms of allocation ratio looks better with increasing thread count. Consequently, if we would progressively increase the assigned threads boundlessly, we would circle back to the single-threaded performance, both for allocation and convergence time (close to saddle-shaped curve).

4.8.1.4 Discussion

Edge computing will play a crucial role in reshaping the future of cloud networks infrastructure. New services will leverage the processing capabilities offered at the network edge for different purposes. However, there are many orthogonal problems currently affecting edge computing which also emerged from the results gathered in our research work. Here, we will discuss two aspects fundamentally connected to QoE delivered to end-users as a function of offloading computation to the edge-cloud infrastructure.

Application & Network. Immersive applications, such as AR/VR, have strict latency requirements as even small delays can results in motion sickness and dizziness. As QoS of network communication technologies (*e.g.*, 5G and millimeter waves) improve (*i.e.*, shorter network delay and higher throughput [227, 228, 229]), it becomes crucial to optimize the utilization of the compute capabilities and task allocation mechanisms at edge.

In spite of that, end-to-end application latency still accounts for the most significant fraction of the perceived user experience, as shown in the results presented in this section and the attached research paper (Publication IV). We focused on task execution time and network latency while ignoring the non-marginal overhead introduced by other components. These additional delays may have many sources, including the operating system, bloated network queues, application logic, network fluctuations (retransmissions, packet loss), to name a few.

With Nimbus, we did not take into account these variables to keep the problem tractable, since added delay caused by some of the above is hardly predictable. Consequently, our results are to be considered an optimistic estimate on top of which application logic and context overhead must be added.

Smartphone evolution. In Nimbus, we looked at possibilities to offload demanding computational steps from mobile devices to the edge infrastructure. However, the ever-increasing computational capacities of smartphones [230, 231, 232], and more general-purpose utility of edge computing demands re-thinking the applicability of edge for mobile clients. For example, high-end smartphones equipped with powerful mobile GPUs benefit more from running computations locally than offloading, due to higher efficiency (in energy consumption and inference time) offered by their processor architectures and algorithms [233]. On the other hand, with many applications competing for the mobile GPU, the performance might decrease due to throttling. Edge resources can be used to further enhance what can be achieved by a smartphone. An example are sophisticated

and accurate neural networks – which are often prohibitive for smartphones as they require considerably more RAM and computational power.

4.9 Summary

In this chapter, we presented our work in two categories of system research for edge computing. We started with a prototype version of the our unikernel orchestration framework, FADES. Then, we addressed some of its limitations in ECCO on top of enabling distributed, edge-cloud execution pipelines. In a distributed environment, migration is crucial to move computation across different nodes to address unforeseen situations such as failures or resource depletion. Therefore, we added the last piece to the puzzle by presenting MirageManager: the first, stateful, unikernel migration system. Finally, we looked at the same problem but at scale and stepped back from implementation-specific challenges. In Nimbus, we crafted a resource provisioning algorithm to manage hardware resources at scale in a distributed, multi-tier, edge-cloud infrastructure. This last piece of research work acts as a wrapper around our orchestration framework by dealing with fundamental question of how to opportunely allocate resources without saturating the infrastructure. In the next chapter, we will conclude the thesis by going over the research questions introduced in Chapter 1 and summarize the contributions presented in the previous chapters.

5 Conclusion & Outlook

This thesis proposed several solutions for the integration of edge and cloud computing towards finding a sweet spot where their interplay can bring benefits to existing services. In this chapter, we summarize our contributions in relation to the problem and research questions posed in Chapter 1. We further discuss limitations of the presented solutions and opportunities for future work.

5.1 Research Questions Summary

RQ1 **What virtualization technique should be employed to match the requirements of both edge and cloud?**

Edge and cloud exhibit different requirements which need to be accommodated in order to make the best out of their interplay. This extends also to virtualization techniques which, as described in Chapters 2 and 4, are extensively used in DCs to orchestrate applications. In our work, we aimed at identifying a solution to virtualize resources at the edge and in the cloud without creating a gap between the two architectures. To answer the research question, we identified unikernels as promising technology designed for the cloud and, as well, very well-suited to match the edge networks requirements. In Section 4.2, we showed the potential of unikernels in comparison to other virtualization techniques such as containers and classical VMs.

RQ2 **What model of computation should be adopted to support the interplay between edge and cloud in order to support different classes of applications?**

As discussed in Chapter 1, one of the challenges for edge computing and distributed systems in general is scalability. In fact, there is a need for solutions where applications are automatically partitioned and distributed in order to maximize or improve the service quality. In Chapter 4, we introduced our unikernel orchestration framework paired with a distributed task (*pipelines*) chaining mechanism. In our computational model, services are split, distributed, and moved towards the data source in contrast to the opposite paradigm typically found with cloud-based applications. Our model of computation allows to re-partition the *compute responsibility* between cloud and edge with the goal of reducing unnecessary uplink data transfers and optimize performance by leveraging spare computational resources found at the edge.

RQ3 **How to support stateful services in fault-prone and resource-constrained edge networks?**

The Internet of Things and edge networks are primarily composed of resource-constrained devices bringing many challenges such as resource provisioning, security, faults and updates management, and so forth. In Chapter 2 and 4 we provided details about this challenges and focused on the problem of supporting stateful applications in an edge infrastructure. In Section 4.7, we introduced one approach based on moving computation from one machine to the next without losing data. Specifically, we presented MirageMigration: a tool to enable stateful migration of unikernels by means of a state checkpointing library. By that, we answered our RQ of how to maintain the state of stateful applications without invasive changes to the underlying virtualization technology Additionally, in Section 3.1 we introduces a unikernel-based application (UIDS) which would benefit from preserving its state migration in order to deliver its monitoring capabilities (e.g., keeping track of malicious connections).

RQ4 **How to allocate resource efficiently in an edge-cloud infrastructure to improve QoE for end-users?**

The last research question addresses the problem of provisioning resources in an edge-cloud infrastructure, at scale. The considerable number of heterogeneous devices at the edge complicates non-trivially the effort required to allocate services and tasks to these devices while guaranteeing specific QoE requirements. With Nimbus, introduced in Section 4.8.1, we tackle this problem especially in the context of real-time applications such as AR — a use case which we also explored in Section 3.2. Our resource provisioning algorithm showed how it is possible to increase the quality of experience of mobile, multimedia applications by following a best-effort approach. By leveraging a multi-layer, edge-cloud infrastructure we allocate mobile users tasks solving a multi-objective optimization problem striking a balance between performance and infrastructure load.

The research work presented in this thesis is a non-comprehensive *excursus* on edge computing covering both system and application challenges in relation to different use-case. There are many lessons to be learned from this research journey which can be re-conducted to the above research questions. For example, we found many reasons why unikernels can be a promising technique to deploy edge applications and enhanced them with extra features (e.g., stateful migration) to show their adaptability to different requirements. However, unikernels should not be considered the *silver bullet* of virtualization technologies, at least for the time being. Depending on the specific application, other solutions (such as Docker containers or classic VM) might be a better fit. The conclusions drawn in this thesis aim at showing the potential of unikernels and elevate them to be considered a valid alternative to existing, consolidated, virtualization technologies.

From a system perspective, we advocate for the interplay between edge and cloud. The shape and level of interconnection between these two infrastructure paradigm might vary but the only way ahead for them is together. Nowadays, we can see how the boundary

between the two is already becoming more and more evanescent, with cloud providers extending the reach of their data centers to the edge with servers deployed in proximity of end users. However, managing such a vast architecture can prove to be challenging. In this thesis, we just scratch the surface of a complex problem such as QoE-aware resource provisioning. Many variables needs to be taken into account of which many are hard to model (e.g., network latency). Hence, another lesson learned in this research journey is that no matter how precise we model a system, there will be always some amount of unpredictability to be accounted for. From an engineering perspective, this is a synonym of aiming at a very good solution, rather than the optimal, as long as all no constraints are violated. This is what we explored with Nimbus as a conclusion of the research effort.

In the next section, we will discuss about the future work that could be conducted to continue this journey and give more insights about the limitations and challenges of our solutions.

5.2 Future Work

We presented several techniques to bridge the gap between the edge and cloud infrastructures. While our solutions were complete in terms of required functionalities, they are still open to improvements. The infrastructure design still requires adjustments and modifications to adapt to different use-cases and scenarios. Our edge-cloud platform is based on unikernels which are still in their infancy and have much less support compared to other well-established solutions such as Docker. When using Docker, a developer just needs to embed their application into the container without no major changes required to the actual service code. What the developer needs to learn is only how to use Docker. With unikernels, there is also the extra burden of *porting* (and, eventually, modifying) the original service code into the unikernels and compiling it against the underlying libOS. Moreover, while unikernels are available in different languages (e.g. OCaml, C++, etc.), their implementation can vary greatly in terms of functionality and maturity of the build engine (e.g., MirageOS, IncludeOS). Libraries which are commonly available for standard OSes require substantial work to be ported and used into an unikernel which is by design built against a minimalist OS. These limitations call for sophisticated tools able to automatically translate or adapt existing software artifacts to be directly used inside an unikernel. For example, this extends also to our framework MirageManager. In that case, the presence of an automated tool to build programmatically the execution graph of the unikernel application logic would greatly reduce the developer effort. On the other hand, unikernels benefit from a behavior defined at compile-time which opens to the possibility of using formal proof management system like Coq [234] and CompCert [235] (verified C compiler). These can help in verifying components of MirageOS (such as the garbage collector), as well as to support hardware compilation to FPGAs for datacenters or new experimental CPU targets such as the BERI processor [236]. Based on this observations, it is important that future research on unikernels is aimed at low-

ering this entry barriers for the developers to foster the adoption of this new technology and help the proliferation of new open-source projects.

In terms of resource management, there is still considerable work to be done in order to build a scalable edge-cloud platform able to serve many users. With Nimbus, we showed how proper allocation of end user tasks can boost performance of real-time applications. However, we left aside some aspects such as security and specifically policies to create a fair environment. For example, we do not restrict an end-user to a maximum time for which they can utilize the edge-cloud infrastructure. This can lead to numerous problems: a malicious user might decide to offload tasks forever and on multiple servers to leech resources from the infrastructure, which may lead to starvation. One solution could be a credit system, where each user can only utilize services offered by the edge-cloud by spending some virtual currency. Naturally, this can be based of largely explored technologies such as blockchain and smart contracts [237]. Other possible approaches could be introducing a fixed time limit after which the user is forcefully rescheduled. However, all these solutions require users to be registered so that system can keep track of their credit or the amount of time spent using the service.

Data and analysis has moved further out to the edge, with a wide range of sensors and monitoring devices gathering information for almost every conceivable purpose. The by-product of this trend is the inevitable temptation to exploit edge devices vulnerabilities, compromise the data, or take over large edge networks to create botnets. Moreover, edge devices are often deployed outside a centralized data infrastructure, making it fundamentally harder to monitor from both a digital and physical security standpoint. As part of our research on edge computing, we also looked into the topic of network security with UIDS: a lightweight IDS deployable at the edge. While our solutions showed great potential, it still falls short in addressing a few challenges. For example, signature-based detectors are static in the sense that in order to detect new attacks an update is necessary. Advances in machine learning make anomaly-based IDSes interesting as future exploration venue due to their flexibility and capability to adapt and learn online new attack vectors. Consequently, one possible future venue of exploration could be extending UIDS with anomaly based detection using machine learning libraries. Eventually, this could be combined with ECCO to create a network of IDSs leveraging federated learning to constantly improve while making use of the distributed resources found at the edge. Finally, IDS are just one component in the security landscape which cannot protect alone an infrastructure. Therefore, further research is required to ensure privacy and security of edge application providers to deliver an effective edge-cloud infrastructure.

Concluding, edge computing is an exciting and promising research field where multiple challenges remain to be addresses. It has received significant attention from the research community to resolve several pressing issues obstructing its real-world adoption. In this thesis, we tackled primarily system research challenges by developing and improving from the ground-up a unikernel orchestration framework addressing many short-comings of current systems. Our work progressively improved with each contribution by adding new fundamentals functionality (e.g., distributed task execution, stateful migration) motivated by concrete use-cases. At the pinnacle of our edge computing journey, we investigated research provisioning issues in large scale, distributed systems which have

a key role in improving end-users experience - one of the crucial selling points of hybrid, edge-cloud infrastructures.

Acronyms

2PMT	Two-Phase Multi Threaded.
3D	Three Dimensional.
ABI	Application Binary Interface.
AP	Access Point.
API	Application Programming Interface.
AR	Augmented Reality.
BBUs	Baseband Units.
BCI	Brain-Computer Interfaces.
BT	binary translation.
C-RAN	Cloud Radio Access Network.
CDN	Content Delivery Networks.
CPU	Central Processing Unit.
CRIU	Checkpoint/Restore In Userspace.
DAG	Directed Acyclic Graph.
DC	Datacenter.
DMF	Data Manipulation Functions.
DoS	Denial of Service.
DRB	Data Resource Broker.
DS	Design Science.
DTO	Data Retrieval Operation.
ECDF	Empirical distribution function.
EF	Edge Function.
EN	Edge Node.
ESes	External Services.
ETSI	European Telecommunications Standards Institute.
FaaS	Function as a service.
FPS	Frames per Second.
GPIO	General-purpose input/output.
GPU	Graphic Processing Unit.

HMD	Head-Mounted Display.
IaaS	Infrastructure as a Service.
ICMP	Internet Control Message Protocol.
ICT	Information and Communication Technology.
IDS	Intrusion Detection System.
IIoT	Industrial Internet of Things.
IOMMU	Input/Output Memory Management Unit.
IoT	Internet of Things.
ISG	Industry Specification Group.
IT	Information Technology.
JVM	Java Virtual Machine.
KVM	Kernel-based Virtual Machine.
LibOS	Library Operating System.
LOIT	Low Orbit Ion Cannon.
LTE	Long Term Evolution.
LV	Lightweight Virtualization.
MCC	Mobile Cloud Computing.
MD	Mobile Device.
MEC	Mobile Edge Computing.
MPC	Multiparty Computation.
MT	Multi Threaded.
MTW	Metadata Task Wrapper.
NFV	Network Function Virtualization.
NIST	National Institute of Standard and Technologies.
OCI	Open Container Initiative.
ORC	Orchestrator.
OS	Operative System.
P2P	Peer-to-Peer.
PaaS	Platform as a Service.
PC	Personal Computer.
PCI	Peripheral Component Interconnect.
PVM	Para-Virtualized Machines.
QEMU	Quick EMUlator.
QoE	Quality of Experience.

QoS	Quality of Service.
RAM	Random Access Memory.
RAN	Radio Access Network.
RAT	Radio Access Technology.
REST	Representational state transfer.
RNC	Radio Network Controller.
RRHs	Radio Remote Heads.
RTT	Round Trip Time.
SaaS	Software as a Service.
SAC	Sensor Access Controller.
SCADA	Supervisory Control and Data Acquisition.
SLA	Service Level Agreement.
SoC	System-on-a-Chip.
ST	Single Threaded.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol).
UI	User Interface.
V2X	Vehicle-to-Everything.
Virtual Reality	VR.
VM	Virtual Machine.
VMM	Virtual Machine Monitor.
VPN	Virtual Private Network.
VT	Intel Virtualization Technology.
WLAN	Wireless Local Access Network.

Bibliography

[1] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jörg Ott. Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111, 2018.

[2] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, page 36–41, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Vittorio Cozzolino, Oliver Flum, Aaron Yi Ding, and Jörg Ott. Miragemanager: Enabling stateful migration for unikernels. In *Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems*, CCIoT '20, page 13–19, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Vittorio Cozzolino, Leonardo Tonetto, Nitinder Mohan, Aaron Yi Ding, and Jörg Ott. Nimbus: A latency-energy efficient edge-cloud allocation algorithm for task offloading. Submitted to IEEE Transaction on Cloud Computing. Under review.

[5] Vittorio Cozzolino, Oleksii Moroz, and Aaron Yi Ding. The virtual factory: Hologram-enabled control and monitoring of industrial iot devices. In *2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pages 120–123, 2018.

[6] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Edge chaining framework for black ice road fingerprinting. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '19, page 42–47, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Vittorio Cozzolino, Nikolai Schwellnus, Jörg Ott, and Aaron Yi Ding. Uids: Unikernel-based intrusion detection system for the internet of things. In *DISS 2020-Workshop on Decentralized IoT Systems and Security*, 2020.

[8] Vittorio Cozzolino, Jörg Ott, Aaron Yi Ding, and Richard Mortier. ECCO: Edge-cloud chaining and orchestration framework for road context assessment. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 223–230, 2020.

[9] IBM 7030 Stretch. `https://en.wikipedia.org/wiki/IBM_7030_Stretch`. Accessed: 2020-09-21.

[10] Neuralink: Elon Musk unveils pig with chip in its brain. `https://www.bbc.com/news/world-us-canada-53956683`. Accessed: 2020-09-21.

[11] Dave Evans. The internet of things how the next evolution of the internet is changing everything (april 2011). *White Paper by Cisco Internet Business Solutions Group (IBSG)*, 2012.

[12] Growing opportunities in the Internet of Things. `https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things`. Accessed: 2020-07-31.

[13] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric computing: Vision and challenges, 2015.

[14] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud security and privacy: an enterprise perspective on risks and compliance.* " O'Reilly Media, Inc.", 2009.

[15] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[16] Koustabh Dolui and Soumya Kanti Datta. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6. IEEE, 2017.

[17] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[18] Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. Iot gateway: Bridgingwireless sensor networks into internet of things. In *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 347–352. Ieee, 2010.

[19] Roberto Morabito, Riccardo Petrolo, Valeria Loscri, and Nathalie Mitton. Legiot: A lightweight edge gateway for the internet of things. *Future Generation Computer Systems*, 81:1–15, 2018.

[20] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377. IEEE, 2018.

[21] Zeyi Tao, Qi Xia, Zijiang Hao, Cheng Li, Lele Ma, Shanhe Yi, and Qun Li. A survey of virtual machine management in edge computing. *Proceedings of the IEEE*, 107(8):1482–1499, 2019.

[22] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[23] Tuyen X Tran and Dario Pompili. Joint task offloading and resource allocation for multi-server mobile-edge computing networks. *IEEE Transactions on Vehicular Technology*, 68(1):856–868, 2018.

[24] Changsheng You, Kaibin Huang, Hyukjin Chae, and Byoung-Hoon Kim. Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications*, 16(3):1397–1411, 2016.

[25] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314, 2011.

[26] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010.

[27] THE NVIDIA EGX PLATFORM FOR EDGE COMPUTING. `https://www.nvidia.com/en-us/data-center/products/egx-edge-computing/`. Accessed: 2020-09-22.

[28] AWS for the Edge. `https://aws.amazon.com/edge/`. Accessed: 2020-09-22.

[29] Google Edge Network. `https://peering.google.com/`. Accessed: 2020-09-22.

[30] Michael Hogan, Fang Liu, Annie Sokol, and Jin Tong. Nist cloud computing standards roadmap. *NIST Special Publication*, 35:6–11, 2011.

[31] Rachit Agarwal, Jen Rexford, et al. Wide-area data analytics. *arXiv preprint arXiv:2006.10188*, 2020.

[32] Jihong Yan, Yue Meng, Lei Lu, and Lin Li. Industrial big data in an industry 4.0 environment: Challenges, schemes, and applications for predictive maintenance. *IEEE Access*, 5:23484–23491, 2017.

[33] Marco S Reis and Geert Gins. Industrial process monitoring in the big data/industry 4.0 era: From detection, to diagnosis, to prognosis. *Processes*, 5(3):35, 2017.

[34] The Increasing Threat to Network Infrastructure Devices and Recommended Mitigations. `https://www.cisa.gov/uscert/ncas/alerts/TA16-250A`. Accessed: 2022-03-19.

[35] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78. IEEE, 2015.

[36] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81, 2014.

[37] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[38] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[39] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[40] Josef Spillner, Johannes MüLler, and Alexander Schill. Creating optimal cloud storage systems. *Future Generation Computer Systems*, 29(4):1062–1072, 2013.

[41] Alessio Botta, Walter De Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: a survey. *Future generation computer systems*, 56:684–700, 2016.

[42] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[43] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[44] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[45] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[46] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[47] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. Serendipity: Enabling remote computing among intermittently connected mobile devices. In *Proceedings of the thirteenth ACM international symposium on Mobile Ad Hoc Networking and Computing*, pages 145–154, 2012.

[48] Weiwen Zhang, Yonggang Wen, Jun Wu, and Hui Li. Toward a unified elastic computing platform for smartphones with cloud support. *IEEE Network*, 27(5):34–40, 2013.

[49] J-P Hubaux, Thomas Gross, J-Y Le Boudec, and Martin Vetterli. Toward self-organized mobile ad hoc networks: the terminodes project. *IEEE Communications Magazine*, 39(1):118–124, 2001.

[50] Abderrahmen Mtibaa, Afnan Fahim, Khaled A Harras, and Mostafa H Ammar. Towards resource sharing in mobile device clouds: Power balancing across mobile devices. *ACM SIGCOMM Computer Communication Review*, 43(4):51–56, 2013.

[51] Abderrahmen Mtibaa, Khaled A Harras, and Afnan Fahim. Towards computational offloading in mobile device clouds. In *2013 IEEE 5th international conference on cloud computing technology and science*, volume 1, pages 331–338. IEEE, 2013.

[52] Wenli Chen, Nitin Jain, and Suresh Singh. Anmp: Ad hoc network management protocol. *IEEE Journal on selected areas in communications*, 17(8):1506–1531, 1999.

[53] Bilel Zaghdoudi, Hella Kaffel-Ben Ayed, and Imen Riabi. Ad hoc cloud as a service: a protocol for setting up an ad hoc cloud over manets. *Procedia Computer Science*, 56:573–579, 2015.

[54] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36, 2012.

[55] Min Chen, Yixue Hao, Yong Li, Chin-Feng Lai, and Di Wu. On the computation offloading at ad hoc cloudlet: architecture and service modes. *IEEE Communications Magazine*, 53(6):18–24, 2015.

[56] Zongqing Lu, Jing Zhao, Yibo Wu, and Guohong Cao. Task allocation for mobile cloud computing in heterogeneous wireless networks. In *2015 24th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2015.

[57] Raquel Lacuesta, Jaime Lloret, Sandra Sendra, and Lourdes Peñalver. Spontaneous ad hoc mobile cloud computing network. *The Scientific World Journal*, 2014, 2014.

[58] Yanmin Gong, Chi Zhang, Yuguang Fang, and Jinyuan Sun. Protecting location privacy for task allocation in ad hoc mobile cloud computing. *IEEE Transactions on Emerging Topics in Computing*, 6(1):110–121, 2015.

[59] Ahmed Hammam and Samah Senbel. A trust management system for ad-hoc mobile clouds. In *2013 8th International Conference on Computer Engineering & Systems (ICCES)*, pages 31–38. IEEE, 2013.

[60] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[61] Jiang Zhu, Douglas S Chan, Mythili Suryanarayana Prabhu, Preethi Natarajan, Hao Hu, and Flavio Bonomi. Improving web sites performance using edge servers in fog computing architecture. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 320–323. IEEE, 2013.

[62] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[63] Tuan Nguyen Gia, Mingzhe Jiang, Amir-Mohammad Rahmani, Tomi Westerlund, Pasi Liljeberg, and Hannu Tenhunen. Fog computing in healthcare internet of things: A case study on ecg feature extraction. In *2015 IEEE international conference on computer and information technology; ubiquitous computing and communications; dependable, autonomic and secure computing; pervasive intelligence and computing*, pages 356–363. IEEE, 2015.

[64] Tanweer Alam. Iot-fog: A communication framework using blockchain in the internet of things. *arXiv preprint arXiv:1904.00226*, 2019.

[65] Mohammad Aazam and Eui-Nam Huh. Fog computing and smart gateway based communication for cloud of things. In *2014 International Conference on Future Internet of Things and Cloud*, pages 464–470. IEEE, 2014.

[66] Cheng Huang, Rongxing Lu, and Kim-Kwang Raymond Choo. Vehicular fog computing: architecture, use case, and security and forensic challenges. *IEEE Communications Magazine*, 55(11):105–111, 2017.

[67] Aleksandra Checko, Henrik L Christiansen, Ying Yan, Lara Scolari, Georgios Kardaras, Michael S Berger, and Lars Dittmann. Cloud ran for mobile networks—a technology overview. *IEEE Communications surveys & tutorials*, 17(1):405–426, 2014.

[68] Jinkun Cheng, Yuanming Shi, Bo Bai, and Wei Chen. Computation offloading in cloud-ran based mobile cloud computing system. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2016.

[69] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE access*, 6:6900–6919, 2017.

[70] Wei Yu, Guobin Xu, Zhijiang Chen, and Paul Moulema. A cloud computing based architecture for cyber security situation awareness. In *2013 iEEE conference on communications and network security (cNS)*, pages 488–492. IEEE, 2013.

[71] Zhijiang Chen, Guobin Xu, Vivek Mahalingam, Linqiang Ge, James Nguyen, Wei Yu, and Chao Lu. A cloud computing based network monitoring and threat detection system for critical infrastructures. *Big Data Research*, 3:10–23, 2016.

[72] A First Inside Look at Pokémon GO Battery Drain. `http://mobileenerlytics.com/a-first-inside-look-at-pokemon-go-battery-drain-you-wont-catch-many-if-your-battery-dies-so-quickly/`. Accessed: 2020-04-02.

[73] Charles L Phillips and Royce D Habor. *Feedback control systems*. Simon & Schuster, Inc., 1995.

[74] Benjamin C Kuo. *Automatic control systems*. Prentice Hall PTR, 1987.

[75] Ronald Cramer, Ivan Bjerre Damgård, et al. *Secure multiparty computation*. Cambridge University Press, 2015.

[76] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[77] Rachit Agarwal, Jen Rexford, and with contributions from numerous workshop attendees. Wide-area data analytics, 2020.

[78] Deep Learning – Past, Present, and Future. `https://www.kdnuggets.com/2017/05/deep-learning-big-deal.html`. Accessed: 2020-09-07.

[79] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[80] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[81] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016.

[82] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75, 2018.

[83] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[84] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[85] Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. The rise of deep learning in drug discovery. *Drug discovery today*, 23(6):1241–1250, 2018.

[86] Maryam M Najafabadi, Flavio Villanustre, Taghi M Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1):1, 2015.

[87] Xue-Wen Chen and Xiaotong Lin. Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525, 2014.

[88] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[89] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[90] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–19, 2019.

[91] Xiaofei Wang, Yiwen Han, Chenyang Wang, Qiyang Zhao, Xu Chen, and Min Chen. In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning. *IEEE Network*, 33(5):156–165, 2019.

[92] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 1–11, 2019.

[93] Yongfeng Qian, Long Hu, Jing Chen, Xin Guan, Mohammad Mehedi Hassan, and Abdulhameed Alelaiwi. Privacy-aware service placement for mobile edge computing via federated learning. *Information Sciences*, 505:562–570, 2019.

[94] Saraju P Mohanty, Uma Choppali, and Elias Kougianos. Everything you wanted to know about smart cities: The internet of things is the backbone. *IEEE Consumer Electronics Magazine*, 5(3):60–70, 2016.

[95] Quan Yuan, Haibo Zhou, Jinglin Li, Zhihan Liu, Fangchun Yang, and Xuemin Sherman Shen. Toward efficient content delivery for automated driving services: An edge computing solution. *IEEE Network*, 32(1):80–86, 2018.

[96] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[97] Tobias Höllerer and Steve Feiner. Mobile augmented reality. *Telegeoinformatics: Location-based computing and services*, 21, 2004.

[98] Ronald Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, and Blair MacIntyre. Recent advances in augmented reality. *IEEE computer graphics and applications*, 21(6):34–47, 2001.

[99] Zhanpeng Huang, Pan Hui, Christoph Peylo, and Dimitris Chatzopoulos. Mobile augmented reality survey: a bottom-up approach. *arXiv preprint arXiv:1309.4413*, 2013.

[100] How is Mobile AR Landing with Consumers? `https://virtualrealitypop.com/how-is-mobile-ar-landing-with-consumers-cbc4b14e5957`. Accessed: 2020-04-27.

[101] Latency – the sine qua non of AR and VR. `http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/`. Accessed: 2020-04-27.

[102] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. Future networking challenges: The case of mobile augmented reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1796–1807. IEEE, 2017.

[103] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[104] Zhanpeng Huang, Weikai Li, Pan Hui, and Christoph Peylo. Cloudridar: A cloud-based architecture for mobile augmented reality. In *Proceedings of the 2014 workshop on Mobile augmented reality and robotic technology-based systems*, pages 29–34, 2014.

[105] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[106] Dimitris Chatzopoulos, Carlos Bermejo, Zhanpeng Huang, and Pan Hui. Mobile augmented reality survey: From where we are to where we go. *Ieee Access*, 5:6917–6950, 2017.

[107] Toward Industry 4.0 With IoT: Optimizing Business Processes in an Evolving Manufacturing Factory. `https://www.frontiersin.org/articles/10.3389/fict.2019.00017/full`. Accessed: 2020-09-08.

[108] Baotong Chen, Jiafu Wan, Antonio Celesti, Di Li, Haider Abbas, and Qin Zhang. Edge computing in iot-based manufacturing. *IEEE Communications Magazine*, 56(9):103–109, 2018.

[109] Pankesh Patel, Muhammad Intizar Ali, and Amit Sheth. On using the intelligent edge for iot analytics. *IEEE Intelligent Systems*, 32(5):64–69, 2017.

[110] Dimitrios Georgakopoulos, Prem Prakash Jayaraman, Maria Fazia, Massimo Villari, and Rajiv Ranjan. Internet of things and edge cloud computing roadmap for manufacturing. *IEEE Cloud Computing*, 3(4):66–73, 2016.

[111] Takuo Suganuma, Takuma Oide, Shinji Kitagami, Kenji Sugawara, and Norio Shiratori. Multiagent-based flexible edge computing architecture for iot. *IEEE Network*, 32(1):16–23, 2018.

[112] Thomas Erl. *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 1900.

[113] What is orchestration? `https://www.redhat.com/en/topics/automation/what-is-orchestration`. Accessed: 2020-09-09.

[114] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135. IEEE, 2015.

[115] Muhammad Alam, Joao Rufino, Joaquim Ferreira, Syed Hassan Ahmed, Nadir Shah, and Yuanfang Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, 2018.

[116] Luciano Baresi, Danilo Filgueira Mendonça, and Martin Garriga. Empowering low-latency applications through a serverless edge computing architecture. In *European Conference on Service-Oriented and Cloud Computing*, pages 196–210. Springer, 2017.

[117] Eyal de Lara, Carolina S Gomes, Steve Langridge, S Hossein Mortazavi, and Meysam Roodi. Hierarchical serverless computing for the mobile edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 109–110. IEEE, 2016.

[118] Lambda@Edge. `http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html`. Accessed: 2020-09-09.

[119] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[120] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE journal on selected areas in communications*, 14(7):1280–1297, 1996.

[121] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library os from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 291–304, 2011.

[122] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[123] Rafael Weingärtner, Gabriel Beims Bräscher, and Carlos Becker Westphall. Cloud resource management: A survey on forecasting and profiling models. *Journal of Network and Computer Applications*, 47:99–106, 2015.

[124] Fangzhe Chang, Jennifer Ren, and Ramesh Viswanathan. Optimal resource allocation in clouds. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 418–425. IEEE, 2010.

[125] Qi-yi Huang and Ting-lei Huang. An optimistic job scheduling strategy based on qos for cloud computing. In *2010 International Conference on Intelligent Computing and Integrated Systems*, pages 673–675. IEEE, 2010.

[126] What risks do iot security issues pose to businesses? `https://blog.avast.com/iot-security-business-risk`. Accessed: 2021-03-05.

[127] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 250–257. IEEE, 2015.

[128] Eduardo K. Viegas, Altair Santin, and Luiz Soares de Oliveira. Toward a reliable anomaly-based intrusion detection in real-world environments. *Computer Networks*, 127, 08 2017.

[129] Iman Sharafaldin, Arash Habibi Lashkari, and Ali Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. pages 108–116, 01 2018.

[130] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[131] Amin Hassanzadeh, Zhaoyan Xu, Radu Stoleru, Guofei Gu, and Michalis Polychronakis. Pride: Practical intrusion detection in resource constrained wireless mesh networks. In *International Conference on Information and Communications Security*, pages 213–228. Springer, 2013.

[132] Fabian Hugelshofer, Paul Smith, David Hutchison, and Nicholas JP Race. Openlids: a lightweight intrusion detection system for wireless mesh networks. In *Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 309–320, 2009.

[133] Ar Kar Kyaw, Yuzhu Chen, and Justin Joseph. Pi-ids: evaluation of open-source intrusion detection systems on raspberry pi 2. In *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, pages 165–170. IEEE, 2015.

[134] A. Varet and N. Larrieu. How to generate realistic network traffic? In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 299–304, July 2014.

[135] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[136] Dominic Gorecky, Mathias Schmitt, Matthias Loskyll, and Detlef Zühlke. Human-machine-interaction in the industry 4.0 era. In *2014 12th IEEE international conference on industrial informatics (INDIN)*, pages 289–294. IEEE, 2014.

[137] Volker Paelke. Augmented reality in the smart factory: Supporting workers in an industry 4.0. environment. In *Proceedings of the 2014 IEEE emerging technology and factory automation (ETFA)*, pages 1–4. IEEE, 2014.

[138] Microsoft Hololens. `https://www.microsoft.com/en-us/hololens`. Accessed: 2020-11-26.

[139] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[140] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[141] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[142] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks*, 53(17):2923–2938, 2009.

[143] Mohsen Tarighi, Seyed A Motamedi, and Saeed Sharifian. A new model for virtual machine migration in virtualized cluster server based on fuzzy decision making. *arXiv preprint arXiv:1002.3329*, 2010.

[144] Emmanuel Arzuaga and David R Kaeli. Quantifying load imbalance on virtualized enterprise servers. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 235–242, 2010.

[145] Liuhua Chen, Haiying Shen, and Karan Sapra. Rial: Resource intensity aware load balancing in clouds. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 1294–1302. IEEE, 2014.

[146] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[147] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. Ieee, 2010.

[148] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *International Workshop on Quality of Service*, pages 381–398. Springer, 2003.

[149] Haiying Shen and Liuhua Chen. Distributed autonomous virtual resource management in datacenters using finite-markov decision process. *IEEE/ACM Transactions on Networking*, 25(6):3836–3849, 2017.

[150] Shridhar G Domanal and G Ram Mohana Reddy. Optimal load balancing in cloud computing by efficient utilization of virtual machines. In *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–4. IEEE, 2014.

[151] Haisheng Tan, Zhenhua Han, Xiang-Yang Li, and Francis CM Lau. Online job dispatching and scheduling in edge-clouds. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[152] Andrew J Younge, Robert Henschel, James T Brown, Gregor Von Laszewski, Judy Qiu, and Geoffrey C Fox. Analysis of virtualization technologies for high performance computing environments. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 9–16. IEEE, 2011.

[153] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, Kang G Shin, et al. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 137, 2007.

[154] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[155] I Parallels. An introduction to os virtualization and parallels virtuozzo containers. *Parallels, Inc, Tech. Rep*, 2010.

[156] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, pages 6–es, 2007.

[157] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *Rpe Report*, 142, 2005.

[158] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[159] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[160] VMware. `https://www.vmware.com/`. Accessed: 2020-29-09.

[161] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008.

[162] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[163] Love H. Seawright and Richard A. MacKinnon. Vm/370—a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[164] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(SI):89–104, 2002.

[165] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[166] Kirill Kolyshkin. Virtualization in linux. *White paper, OpenVZ*, 3(39):8, 2006.

[167] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[168] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[169] T Jones. Linux virtualization and pci passthrough. *developerWorks, IBM Corporation*, 2009.

[170] Production-Grade Container Orchestration. `https://kubernetes.io/`. Accessed: 2022-03-19.

[171] Docker. `https://www.docker.com/`. Accessed: 2020-29-09.

[172] Linux Conainers (LXC). `https://linuxcontainers.org/`. Accessed: 2020-29-09.

[173] What's the Difference Between Containers and Virtual Machines? `https://www.electronicdesign.com/technologies/dev-tools/article/21801722/whats-the-difference-between-containers-and-virtual-machines`. Accessed: 2020-29-09.

[174] David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-quite-so-broken {TLS}: Lessons in re-engineering a security protocol specification and implementation. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 223–238, 2015.

[175] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30–44, 2013.

[176] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 559–573, 2015.

[177] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-wesley professional, 2011.

[178] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications surveys & tutorials*, 18(1):236–262, 2015.

[179] HaLVM. `https://galois.com/project/halvm/`. Accessed: 2020-05-10.

[180] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 459–473, 2014.

[181] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[182] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 61–72, 2014.

[183] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[184] Kubeless. `https://kubeless.io/`. Accessed: 2020-05-10.

[185] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[186] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[187] Serverless, just code focus on what's most important. `https://cloud.google.com/appengine`. Accessed: 2022-01-06.

[188] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[189] Wikipedia contributors. Electronic waste — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Electronic_waste&oldid=1005349759`, 2021. [Online; accessed 18-February-2021].

[190] A new circular vision for electronics, time for a global reboot. `https://www.weforum.org/reports/a-new-circular-vision-for-electronics-time-for-a-global-reboot`. Accessed: 2021-02-18.

[191] Latif U Khan, Ibrar Yaqoob, Nguyen H Tran, SM Ahsan Kazmi, Tri Nguyen Dang, and Choong Seon Hong. Edge-computing-enabled smart cities: A comprehensive survey. *IEEE Internet of Things Journal*, 7(10):10200–10232, 2020.

[192] Michael Doering. High-resolution large-scale air pollution monitoring: approaches and challenges. In *Proceedings of the 3rd ACM international workshop on MobiArch*, pages 5–10, 2011.

[193] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 15. ACM, 2017.

[194] Benchmarking Hardware for CNN Inference in 2018. `https://towardsdatascience.com/benchmarking-hardware-for-cnn-inference-in-2018-1d58268de12a`. Accessed: 2020-04-27.

[195] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135, 2015.

[196] Onur Sahin and Ayse K Coskun. Providing sustainable performance in thermally constrained mobile devices. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 72–77, 2016.

[197] Michael B Lewis and Andrew J Edmonds. Face detection: Mapping human performance. *Perception*, 32(8):903–920, 2003.

[198] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. Cloud gaming: architecture and performance. *IEEE network*, 27(4):16–21, 2013.

[199] The future of cloud gaming is on the edge. `https://www.datacenterdynamics.com/en/opinions/future-cloud-gaming-edge/#:~:text=The%20Edge%20enables%20new%20cloud,rendering%20of%20graphically%20intensive%20video`. Accessed: 2021-02-18.

[200] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 26(4):178–183, 2019.

[201] Gopika Premsankar, Mario Di Francesco, and Tarik Taleb. Edge computing for the internet of things: A case study. *IEEE Internet of Things Journal*, 5(2):1275–1284, 2018.

[202] Sultan Basudan, Xiaodong Lin, and Karthik Sankaranarayanan. A privacy-preserving vehicular crowdsensing-based road surface condition monitoring system using fog computing. *IEEE Internet of Things Journal*, 4(3):772–782, 2017.

[203] Andrew J Wixted, Peter Kinnaird, Hadi Larijani, Alan Tait, Ali Ahmadinia, and Niall Strachan. Evaluation of LoRa and LoRaWAN for wireless sensor networks. In *2016 IEEE SENSORS*, pages 1–3. IEEE, 2016.

[204] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. *IEEE Transactions on Vehicular Technology*, 65(6):3860–3873, 2016.

[205] Yunxin Jeff Li. An overview of the DSRC/WAVE technology. In *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*, pages 544–558. Springer, 2010.

[206] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[207] Wikipedia contributors. Topological sorting — Wikipedia, the free encyclopedia, 2020. [Online; accessed 20-October-2020].

[208] Reza Olfati-Saber, J Alex Fax, and Richard M Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, 2007.

[209] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[210] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[211] Keith S Decker. Distributed problem-solving techniques: A survey. *IEEE transactions on systems, man, and cybernetics*, 17(5):729–740, 1987.

[212] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial intelligence*, 26(3):251–321, 1985.

[213] Robert Wesson, Frederick Hayes-Roth, John W Burge, Cathleen Stasz, and Carl A Sunshine. Network structures for distributed situation assessment. *IEEE Transactions on systems, man, and cybernetics*, 11(1):5–23, 1981.

[214] Thomas C Schelling. *The Strategy of Conflict: With a New Preface by The Author*. Harvard university press, 1980.

[215] Stewart L Tubbs. A systems approach to small group interaction. 2012.

[216] PINCTRL (PIN CONTROL) subsystem. `https://01.org/linuxgraphics/gfx-docs/drm/driver-api/pinctl.html`. Accessed: 2021-03-05.

[217] Jetson TX2 Series Pin and Function Names Guide Application Note. `http://developer.nvidia.com/embedded/dlc/tx2-series-pin-function-names-guide-note`. Accessed: 2021-03-05.

[218] Yuang Chen and Thomas Kunz. Performance evaluation of iot protocols under a constrained wireless access network. In *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*, pages 1–7. IEEE, 2016.

[219] MiniOS. `https://wiki.xenproject.org/wiki/Mini-OS`. Accessed: 2020-06-16.

[220] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[221] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 258–269, 2016.

[222] Susanna Epp. *Discrete mathematics with applications*. Nelson Education, 2010.

[223] Express framework. `https://expressjs.com/`. Accessed: 2020-06-16.

[224] Rumprun. `https://github.com/rumpkernel/rumprun`. Accessed: 2021-03-05.

[225] Valeria Cardellini, Michele Colajanni, and Philip S Yu. Dynamic load balancing on web-server systems. *IEEE Internet computing*, 3(3):28–39, 1999.

[226] Kaisa Miettinen. Introduction to multiobjective optimization: Noninteractive approaches. In *Multiobjective optimization*, pages 1–26. Springer, 2008.

[227] Theodore S Rappaport, Shu Sun, Rimma Mayzus, Hang Zhao, Yaniv Azar, Kevin Wang, George N Wong, Jocelyn K Schulz, Mathew Samimi, and Felix Gutierrez. Millimeter wave mobile communications for 5g cellular: It will work! *IEEE access*, 1:335–349, 2013.

[228] R. Trivisonno, R. Guerzoni, I. Vaishnavi, and D. Soldani. Towards zero latency software defined 5g networks. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 2566–2571, 2015.

[229] J. Sachs, G. Wikstrom, T. Dudda, R. Baldemair, and K. Kittichokechai. 5g radio network design for ultra-reliable low-latency communication. *IEEE Network*, 32(2):24–31, 2018.

[230] Your Phone Is Now More Powerful Than Your PC. `https://insights.samsung.com/2018/08/09/your-phone-is-now-more-powerful-than-your-pc/`. Accessed: 2020-05-22.

[231] How the computing power in a smartphone compares to supercomputers past and present. `https://www.businessinsider.com/infographic-how-computing-power-has-changed-over-time-2017-11?r=DE&IR=T`. Accessed: 2020-05-22.

[232] Mahadev Satyanarayanan, Nathan Beckmann, Grace A. Lewis, and Brandon Lucia. The role of edge offload for hardware-accelerated mobile devices. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile '21, page 22–29, New York, NY, USA, 2021. Association for Computing Machinery.

[233] UbiSpark project. `https://ubispark.cs.helsinki.fi/`. Accessed: 2020-06-30.

[234] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[235] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[236] Robert NM Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A Theodore Markettos, Simon W Moore, Steven J Murdoch, Peter G Neumann, Robert Norton, et al. Bluespec extensible risc implementation: Beri hardware reference. Technical report, University of Cambridge, Computer Laboratory, 2015.

[237] Alexandru Stanciu. Blockchain based distributed control system for edge computing. In *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, pages 667–671. IEEE, 2017.

Publication I

R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar and J. Ott, "Consolidate IoT Edge Computing with Lightweight Virtualization" in IEEE Network, vol. 32, no. 1, pp. 102-111, Jan.-Feb. 2018, doi: 10.1109/MNET.2018.1700175.

Publication Summary

We looked at how Lightweight virtualization (LV) technologies have refashioned the world of software development by introducing flexibility and new ways of managing and distributing software. Today, edge computing complements today's powerful centralized data centers with a large number of distributed nodes that provide virtualization close to the data source and end users. This emerging paradigm offers ubiquitous processing capabilities on a wide range of heterogeneous hardware characterized by different processing power and energy availability.

In this article, we present an in-depth analysis on the requirements of edge computing from the perspective of three selected use cases that are particularly interesting for harnessing the power of the Internet of Things. We discuss and compare the applicability of two LV technologies, containers and unikernels, as platforms for enabling the scalability, security, and manageability required by such pervasive applications that soon may be part of our everyday lives. Additionally, we identify open problems and highlight future directions to serve as a road map for both industry and academia.

Author's Contribution

The paper idea originated with Roberto Morabito, Aaron Yi Ding, and I. Roberto Morabito and I equally contributed to the content of the article. I contributed to the formulation of the problem and discussed the applicability of a specific form of Lightweight Virtualization to a set of application scenarios. In particular, I was involved in the analysis of how unikernels can be applied to sensor data processing in IoT scenarios and for real-time applications. The discussion section was a shared work between all the authors and I specifically focused on elasticity in service provisioning and application portability.

Consolidate IoT Edge Computing with Lightweight Virtualization

Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jörg Ott

***Abstract*—Lightweight Virtualization (LV) technologies have refashioned the world of software development by introducing flexibility and new ways of managing and distributing software. Edge computing complements today's powerful centralized data centers with a large number of distributed nodes that provide virtualization close to the data source and end users. This emerging paradigm offers ubiquitous processing capabilities on a wide range of heterogeneous hardware characterized by different processing power and energy availability. The scope of this article is to present an in-depth analysis on the requirements of edge computing in the perspective of three selected use cases particularly interesting for harnessing the power of the Internet of Things (IoT). We discuss and compare the applicability of two LV technologies, containers and unikernels, as platforms for enabling scalability, security and manageability required by such pervasive applications that soon may be part of our everyday life. To inspire further research, we identify open problems and highlight future directions to serve as a road map for both industry and academia.**

***Index Terms*—IoT, Edge Computing, Container, Unikernel**

I. Introduction

OVER the last decade, the development of the Internet of Things (IoT) has been uphold by the cloud-based infrastructures that aim to cope with the increasing number of IoT services provided by various connected devices. From the initial design, IoT was conceived as extending the Internet with a new class of devices and use cases [1]. This has obviously generated an intrinsic association between IoT and cloud, where the cloud-based network infrastructures are optimized to support a multitude of IoT-centric operations such as service management, computation offloading, data storage, and offline analysis of data.

However, this notion of cloud-connected IoT deployment assumes that most IoT edge networks need to be connected to the cloud, e.g., through some edge gateway and tunnel approach. This centralized model has been challenged recently for meeting the more and more stringent performance requirements of IoT services, especially in terms of latency and bandwidth. In specific, the existing model is not suitable when: a) IoT edge networks create data that needs to be accessed and processed locally, b) piping everything to the cloud and back is not acceptable under delay constraints, and c) the amount of data is too large to transfer to the cloud (in real-time) without causing congestion on the backhaul. Clearly, the highly fragmented and heterogeneous IoT landscape needs to encompass novel and reactive approaches for dealing with these challenges.

R. Morabito, N. Beijar are with Ericsson, Jorvas, Finland. E-Mail: roberto.morabito@ericsson.com, nicklas.beijar@ericsson.com

V. Cozzolino, A. Y. Ding and J. Ott are with Technical University of Munich, Munich, Germany. E-Mail: cozzolin@in.tum.de, ding@in.tum.de, ott@in.tum.de

R. Morabito and V. Cozzolino made equal contributions to this article.

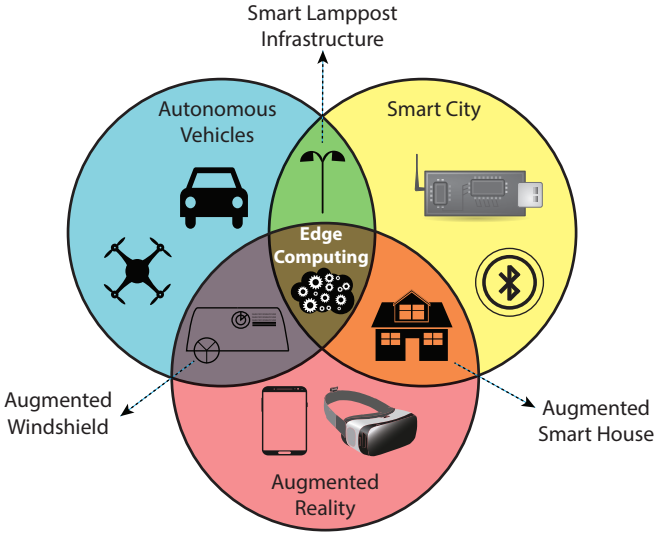


Figure 1: A subset of use cases and services enabled by IoT edge computing

One emerging paradigm, edge computing, represents a new trend to improve the overall infrastructure efficiency by delivering low-latency, bandwidth-efficient and resilient services to IoT users. Although this new approach is not intended to replace the cloud-based infrastructure, it expands the cloud by increasing the computing and storage resources available at the network edge. One typical example is IoT edge offloading [2], which revisits the conventional cloud-based computation offloading where mobile devices resort to resourceful servers to handle heavy computation [3]. To cater for the demands of new IoT services, the computation is reversely dispatched by the servers to constrained devices deployed at the network edge, close to users and data generators.

By harnessing the power of distributed edge resources, the IoT edge computing model can support novel service scenarios such as, for example, autonomous vehicles/drones, smart cities infrastructure and augmented reality (AR). As highlighted in Figure 1, these three representative domains intersect with each other. Edge computing is the linking knot that helps spawn and promote appealing joint services.

Concerning the key aspects of edge computing including scalability, multi-tenancy, security, privacy and flexibility, the fast evolving lightweight virtualization technologies (discussed

in Section IV) have been sorted to fulfill the requirements given their matching features. Meanwhile, we still lack comprehensive guidelines to illustrate how can we exploit the full potential of lightweight virtualization to enhance edge computing, especially for those pleading IoT use cases.

As a solid step towards realizing the IoT edge computing vision, we aim to answer through this article a major question: *Can Lightweight Virtualization (LV), in its different flavors, be exploited for empowering edge architectures and be suitable in a wide range of IoT pervasive environments?* Our use-case study, comparison analysis, and prospect outlook further address the following questions:

- Which LV features can match the increasingly strict requirements of IoT services in constrained environments?
- How can LV and IoT edge scenarios be efficiently utilized together?
- Which challenges must be tackled to effectively exploit the benefits introduced by LV in this context?

The remainder of this article is organized as follows. Motivations of the proposed work are presented in Section II. Section III introduces first the requirements that different *Edge for IoT* cases entail, and then the suitability of LV on mitigating and satisfying them. We introduce LV technologies and illustrate three specific use cases in Section IV and V. Finally, we unveil the open issues and challenges before concluding the article.

II. Motivation

In the context of IoT, edge computing introduces an intermediate layer in the conventional IoT-Cloud computing model. The envisioned edge-driven IoT environment consists of three components: IoT devices, edge layer, and cloud backend. Being a central part of the ecosystem, the edge layer owns the crucial role of bridging and interfacing the central cloud with IoT. Essentially, an edge element in this layer can be characterized by a small to medium size computing entity that aims to provide extra computing, storage, and networking resources to the applications deployed across IoT devices, edge and cloud. Depending on the specific scenario, its functionalities can be executed in cellular base stations, IoT gateways, or more generally, low-power nodes and small data-centers. These may be owned and operated by the user, by a cloud provider or a telecom operator (in Mobile Edge Computing).

Although the placement of a “middle layer" between the end devices and cloud is an architectural concept that is widely utilized in common network infrastructures, such conventional middle layer targets mainly connectivity, routing, and network-oriented functionalities. For example, Network Function Virtualization (NFV) [4], [5] virtualizes typical network elements, such as firewalls, network address translators, switches, and core network components.

For IoT ecosystems, edge computing aims to meet IoT service providers' demand of owning a dedicated infrastructure that is independent of a given technology or use case, and which is capable of satisfy the demanding IoT services' performance requirements. More importantly, in contrary to the plain middle layer solutions, the IoT-centric edge computing must entail programmability and flexibility to deliver ubiquitous processing capabilities across a wide range of heterogeneous hardware. For instance, besides managing IoT home network, the edge layer can simultaneously provide image processing for home camera and data pre-processing operations.

Obviously, the heterogeneous characteristics of various instances and applications deployed on top of the edge layer will generate unique challenges that need to be addressed. From the architectural perspective, this implies that edge layer has to efficiently and mutually cooperate both with cloud-based services and IoT devices, by acting as a bridge between elements that require distinct way of interaction.

In this context, it is crucial to equip the edge layer with tools that allow a flexible, performing, and automated way of efficient services provisioning. Hence, edge elements have to embed service provisioning methods that are independent of the managed applications and communication patterns, and at the same time suitable to different types of traffic and to the application needs, through a cross-layer support. The key is to ensure a virtuous trade-off between design requirements, specific performance targets, and applications manageability spanning the entire three-tier IoT edge computing architecture.

III. Empowering IoT Edge Computing with LV

To fully attain the potential of edge computing for IoT, we need to address four concerns: abstraction, programmability interoperability, and elasticity. In particular for the three-tier IoT edge computing architecture, it is crucial to provide simple and yet efficient configuration and instantiation methods that are independent of the technologies used by different IoT and cloud providers. The tools embedded in edge layer should share common functionalities, exploit common APIs for orchestrating interconnections different networking technologies.

To help us acquire a synoptic view, we highlight the dominant requirements of representative use cases in Figure 2, which encompasses scalability, multi-tenancy, privacy & security, latency, and extensibility.

Compared to alternative virtualization solutions such as hypervisors, we envision a trend towards using lightweight virtualization (LV) technologies in the IoT edge computing. These emerging software solutions can provide the needed supports in terms of hardware abstraction, programmability interoperability, and elasticity. A direct benefit that emerges from employing LV in the IoT edge domain is by avoiding the strict dependency on any given technology or use case. Within a lightweight virtualized instance, either container or unikernel (discussed in Section IV), we can efficiently deploy applications designed to manage and use extremely different technologies. In addition, equipping edge elements with newer services will be made easier since we only need to configure and instantiate stand-alone virtualized applications. This feature avoids complex re-programming and updating operations that are part of the software lifecycle management. Through LV, such complexity is circumvented because updating a particular service requires changes only within a specific virtualized instance.

Scenarios	Requirements				
	Scalability	Multi-tenancy	Privacy & Security	Latency	Extensibility (Open API)
Autonomus Vehicles	*Non critical*	*Non critical*	*Critical. Autonomous vehicles possess sensitive information about the user. Moreover, the constant need of sensors data for navigation make cars a primary target for malicious users.*	*Critical. Cars have strict real-time requirements*	*Non critical. Each car manufacturer will probably run exclusively their own software to ensure security and reliability.*
Augmented Reality	*Critical*	*Critical*	*Critical when processing sensitive multimedia streams.*	*Critical. AR applications require real-time information feed to ensure a smooth and acceptable experience.*	*Critical. Open API are important in this case to enable new services and features.*
Smart Sensors Networks	*Critical*	*Critical due to the number of potential users.*	*Depends on the specific Smart context (for Smart Health it's critical but not for Smart Environment). Strict control over which data can be public is required.*	*Depends. For example, in the case of Machine Type Communications (MTC) it's critical.*	*Critical to enable the creation of an 'IoT Marketplace' where developers can offer new and innovative application exploiting collected data.*
Smart Grid	*Non critical. Data and messages are exchanged at a fixed, predefined rate.*	*Non critical. The infrastructure is usually controlled by a single provider.*	*Critical. Disclosure and analysis of energy consumption information can lead to user profiling and tracking.*	*Critical especially for messages as Phasor Measurement Unit (PMU) or Advanced Metering Infrastructure (AMI).*	*Non critical*
E-Health	*Critical. IoT healthcare networks must be able to meet the growing demand of services from both individuals and health organizations.*	*Critical as multiple healthcare organizations and/or heterogeneous IoT medical devices could share the same network infrastructure.*	*Critical. IoT-edge medical devices deal with personal heath data, which need to be securely stored. Integrity, privacy, and confidentiality must be kept.*	*Depends. It's critical in use-cases as remote surgery. Nevertheless, response time can be acceptable in other scenarios.*	*Critical to support new application able to offer a more accurate patients health condition monitoring.*
Distributed Surveillance	*Critical. Several control units are needed in order to grant the system of better usability and robustness.*	*Non-critical. A single provider usually controls the infrastructure.*	*Critical considering the sensitive information handled.*	*Critical to promptly identify suspects or recognize on-going crimes.*	*Non-critical. Same as Autonomous vehicles.*
Big Data Analytics	*Critical. A big data analytics system must be able to support very large datasets. All the components must be scalable to accommodate the constantly growing amount of data to be handled.*	*Critical. A single Big Data system has to be able to co-locate different use cases, applications, or data sets.*	*Critical. Users share large amount of personal data and sensitive content through their personal devices towards applications (e.g., social networks) and public clouds. Equipping Big Data systems of secure frameworks capable to store and manage user data with high sensitiveness represents a critical aspect.*	*Non critical*	*Critical to improve and deploy different algorithms and tools.*
Network Function Virtualization (NFV)	*Critical. Demand of new services is high and constantly growing.*	*Critical. Resources are shared among customers. A large number of multi-tenant networks run over a physical network.*	*Critical. The use of additional software (e.g., hypervisors, containers or unikernels) extends the chain of trust. Resource pooling and multi-tenancy bring further security/privacy threats.*	*Critical. NFV need to leverage real-time delivery services. NFV introduces additional sources of latency through the virtualization layer.*	*Non critical*

Figure 2: Example of Edge-IoT scenarios requirements

To foster integration with the cloud, LV can also enable cross-platform deployment, allowing a common execution environment across cloud, edge elements, and even constrained IoT devices. The cross-platform deployment benefit introduced by LV further allows both cloud and edge, regardless of their computational hardware capability, to "speak the same language". As suggested in [2], using the same LV instance will enable us to efficiently run them both at the edge and in the cloud, hence achieving a decentralized IoT edge service provisioning architecture. This consequently meets the strict performance requirements of demanding IoT scenarios, and further ensures the crucial requirement of multi-tenancy.

We also note that there are scenarios where virtualization technology is not a suitable option, for manifold reasons. In general, virtualization entails additional delay and resources utilization, which can be challenging for certain real-time or mission-critical tasks that demand low and predictable latency. Moreover, there are fundamental hardware requirements to run a virtualized environment (e.g. a CPU with specific architectural features) that are not easily found on low-end IoT and edge devices.

IV. Overview of Lightweight Virtualization

System virtualization has drastically evolved in the last years offering system architects and developers a plethora of tools to exploit. Therefore, understanding how and when to utilize a specific technology based on the hardware constraints and applicative requirements is a crucial step of the system design phase. Shifting our focus on edge computing and IoT, we identify two main candidates that could address the challenges unique to this domain: containers and unikernels.

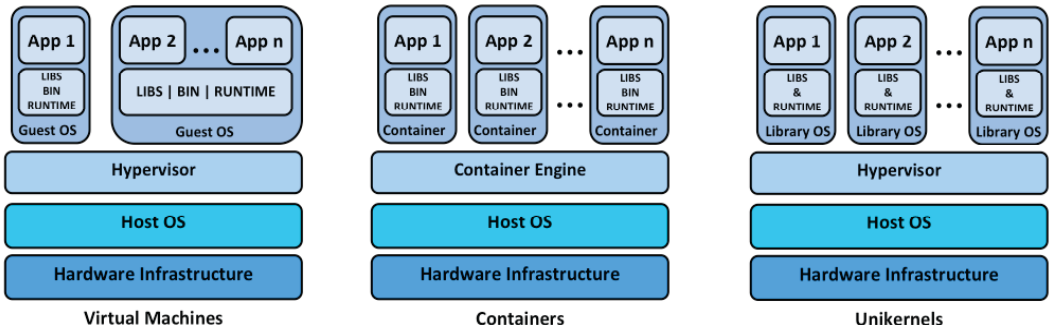
Figure 3 presents both quantitative metrics and architectural differences between the aforementioned technologies, highlighting their main characteristics.

A. Container-based Virtualization: Docker

Container-based virtualization provides a different level of abstraction in terms of virtualization and isolation when compared to other virtualization solutions. In particular, containers can be considered as one of the lightweight alternatives to hypervisor-based virtualization. The conventional hypervisor-based virtualization has been the *de facto* technology used during the last decade for implementing server virtualization and isolation. Hypervisors operate at the hardware level — that is, building customizable virtual hardware and virtual device drivers — thus supporting standalone Virtual Machines (VMs) that are independent and isolated from the underlying host system. In each VM instance, a full Operating System (OS) is typically installed on top of the virtualized hardware, thus generating large VMs images. Furthermore, the emulation of

LV Technique	Property					
	Instantiation time	Image size	Memory footprint	Programming language dependency	Hardware portability	Live migration support
Virtual Machine • KVM • QEMU	*~5/10 secs*	*~1000 MBs*	*~100 MBs*	*No*	*High*	*Yes*
Container • Docker (http://www.docker.com/) • rkt (https://coreos.com/rkt) • OpenVZ https://openvz.org/ • LXC https://linuxcontainers.org/	*~800/1000 msecs*	*~50 MBs*	*~5 MBs*	*No*	*High*	*No*
Unikernel • MirageOS (https://mirage.io/) • HaLVM (http://galois.com/project/) • IncludeOS (www.includeos.org) • ClickOS (http://cnp.neclab.eu/clickos/) • OSv (osv.io)	*~< 50 msecs*	*~< 5MBs (bundle)*	*~8 MBs*	*Yes (i.e., MirageOS unikernels can only be written in OCaml)*	*High*	*No. Requires manual implementation*

(a)



(b)

Figure 3: LV techniques comparison. (a) Quantitative analysis; (b) Core architectural differences

virtual hardware devices and related drivers produces non-negligible performance overhead.

Differently, containers implement processes isolation at the OS level, thus avoiding the virtualization of hardware and drivers [6]. In specific, containers share the same OS kernel with the underlying host machine, meanwhile making it possible to isolate stand-alone applications that own independent characteristics, i.e., independent virtual network interfaces, independent process space, and separate file systems. This shared kernel feature allows containers to achieve a higher density of virtualized instances on a single machine thanks to the reduced image volume.

Containers have achieved much more relevance and practical use recently with the advent of Docker, a high-level platform that has made containers very popular in a short time frame. Docker introduces an underlying *container engine*, together with a practical and versatile API, which allows easily building, running, managing, and removing containerized applications. A *Docker container*, which is a runnable instance of *Docker image*, uses a base image stored in specific private or public registries. Docker uses an overlay file-system (*UnionFS*) to add a read-write layer on top of the image. UnionFS allows to store Docker images as a series of layers and consequently saving disk space. In fact, the different image layers can be cached in the disk allowing to speed up the building process, and re-use the same cached layer for the building of different images.

The lightweight features embedded in containers ease the integration of such technology in various networking fields. In specific to IoT edge computing, containers can enable us to efficiently run containerized applications even in devices characterized by lower processing capabilities, such as Single-Board Computers [7].

B. Library Operating Systems: Unikernels

Unikernels are single-purpose appliances that are at compile time specialized into standalone kernels [8], and sealed against modification after deployment. The concept of unikernels has emerged from the observation that most applications running in the cloud do not require many of the services coming with common operating systems. Additionally, unikernels provide increased security through a reduced attack surface and better performance by dropping unnecessary components from the applications.

Unikernels were designed initially with the cloud in mind but their small footprint and flexibility make them fit also well with the upcoming IoT edge ecosystem as illustrated through different research attempts [9]-[2]. The main differences among existing unikernel implementations sprout from the underlying programming language in use. MirageOS [8] and HaLVM are unikernels based on functional languages with pervasive type-safety in the running code. Other solutions like IncludeOS and ClickOS are C++ unikernels; the former offering a C++ platform to which bind generic applications, while the latter is highly specializing in offering dynamic network processing (based on Click modular router). OSv is based on Java and therefore heavier than the others, but more flexible.

Security and unikernels are tightly coupled. The attack surface of a unikernel is strictly confined to the application

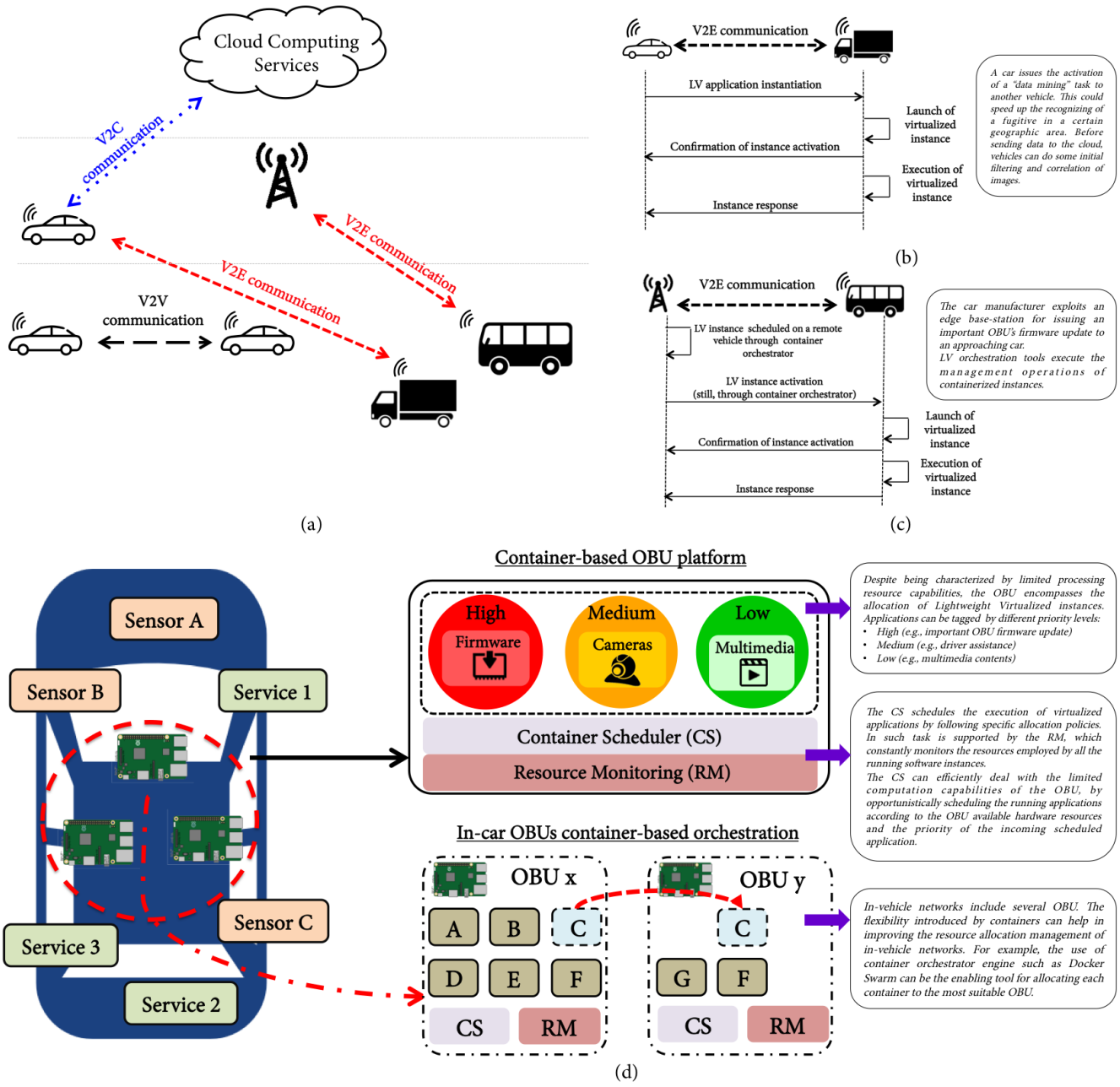


Figure 4: a) Vehicular Edge Computing scenario in its entirety. Vehicular-to-Edge (V2E) interactions examples: b) Car-to-Car V2E communication, c) and Base Station-to-Car V2E communication; d) Container-based virtualization is used for an easier OBU's customization. Furthermore, within the same vehicle orchestration tools are exploited for task offloading among different OBUs.

embedded within. There is no uniform operating layer in a unikernel, and everything is directly compiled into the application layer. Therefore, each unikernel may have a different set of vulnerabilities, which implies that an exploit that can penetrate one may not be threatening the others. Unikernels are principally designed to be stateless. Therefore, they are perfect to embed generally algorithms (e.g. compression, encryption, data aggregation functions) or NFV.

V. Use-case Scenarios

In this section, we present three use cases matching the scenarios presented in Figure 1. Additionally, we illustrate the reasons for adopting a specific LV technology for each case.

A. Towards the Vehicular Edge Computing

The importance of virtualization in vehicular scenarios has been widely acknowledged in the past. Vehicular Cloud-Computing (VCC) represents an efficient architectural model in supporting the *Internet of Vehicles* (IoV) [10]. However, we

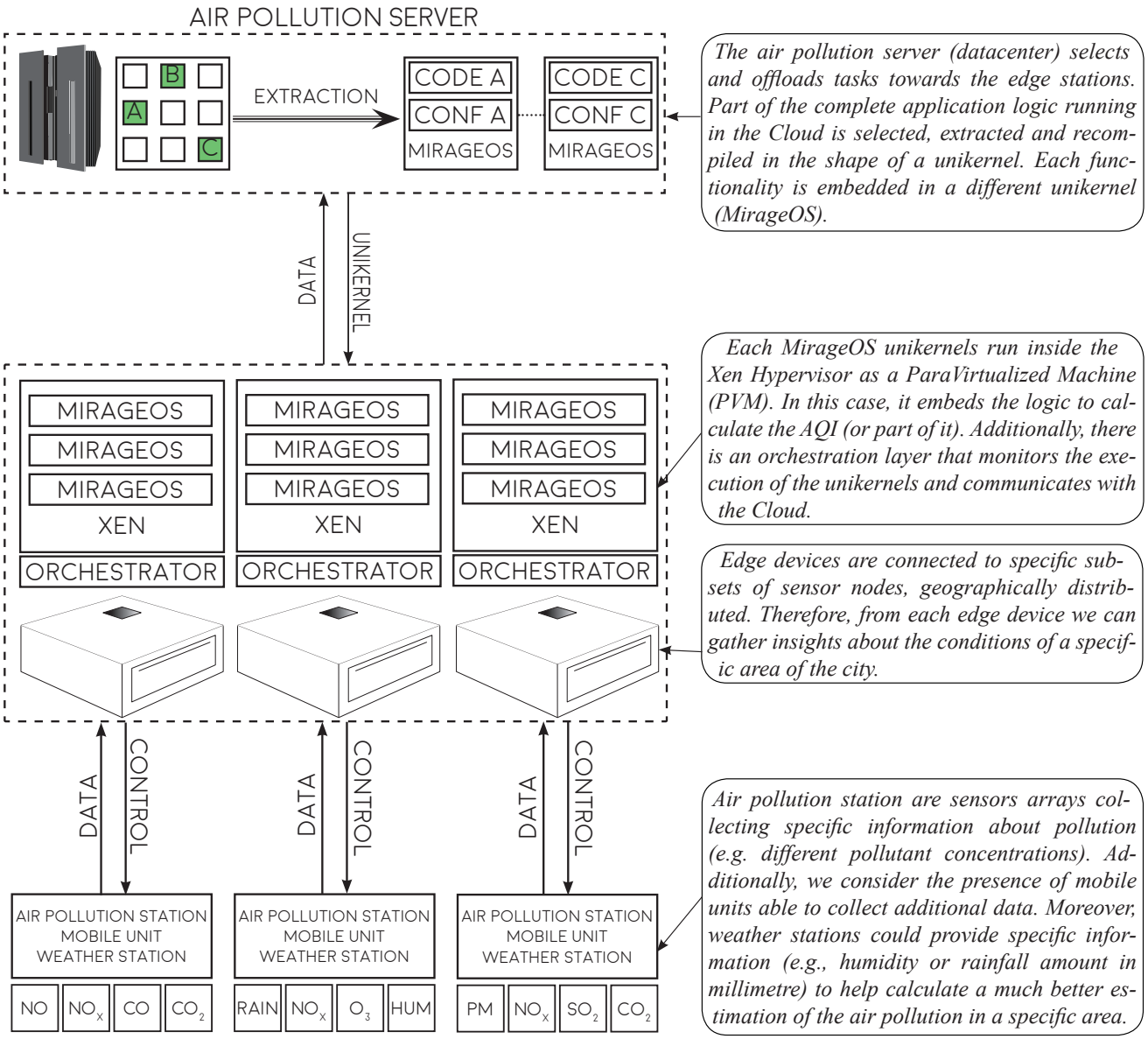


Figure 5: Air pollution scenario. An offloadable task is Air Quality Index (AQI) calculation, a number used by government agencies to communicate to the public how polluted the air currently is or how polluted it is forecast to become. Calculation of the AQI can be executed locally by edge nodes enhancing real-time monitoring.

envision the need to establish a Vehicular Edge Computing (VEC) paradigm, which will play a crucial role in future development of more efficient Vehicle-to-Everything (V2X) systems. VEC can cope with the increasingly strict requirements of V2X applications, and will rely on the growing processing capabilities that the different actors of IoV encompass, including cars' On-Board Unit (OBU), Edge Elements (EEs), Cloud Services. In VEC environments, various units can play the role of EE. Base stations, IoT gateways, and other vehicles themselves can operate as EE by executing specific tasks e.g., lightweight data mining operations, generic off-loading processing, dashcam images filtering, etc. In such context, LV can enable the VEC paradigm, and be exploited in multiple scenarios, spanning from an efficient and flexible customization of cars' OBU to Vehicular-to-Edge (V2E) interactions.

Figure 4 depicts the VEC scenario in its entirety (Fig. 4a), together with practical examples of the way in which LV can be employed in V2E Interactions (Fig. 4b-c) and distributed In-Car Platforms (Fig. 4d).

V2E Interactions. Differently from already well-established Vehicular-to-Vehicular (V2V) communication, V2E aims to encompass computation offloading, tasks outsourcing, and software management operations. In practice, LV-enabled OBU can execute a specific task issued by another vehicle or any other EEs, and vice versa, as shown in the two examples shown in Fig. 4b-c.

In-Car Platforms. Container-based virtualization can be used for OBUs customization. It offers high flexibility in the

platform's software management, and allows overcoming the complex software updating procedures required by OBUs [11]. Through conventional VM, car manufacturers can access all CAN (Controller Area Network) bus sensors through OBD and dashcam. However, given OBUs are embedded systems with limited computational resources, LV's lightweight features avoid the performance overhead and allow scaling up/down the running applications according to specific priorities. Furthermore, by taking into consideration that several OBUs can be distributed within a car, virtualization orchestration tools can be used for OBUs' tasks outsourcing — still by following specific OBU resource management policies. More details about the usage of LV in In-Car platforms can be found in Fig. 4d

B. Edge Computing for Smart City

In the context of Smart City, the measurement of environmental data has become an important issue especially for highly crowded metropolis. Currently, air pollution monitoring is achieved with a sparse deployment of stationary, expensive measurement units embedding both sensors[1] and computing units. Air pollution is predicted based on the measured data in combination with complex mathematical models [12]. Since the cost of deploying and maintaining such pollution station are often prohibitive, we envision crowd-sensing as a tangible solution that combines LV and edge computing.

Edge computing offers resources close to the crowd-sensing entities, which can offload through a direct connection their collected data without using a mobile connection. LV allows to offload, distribute and execute part of the required mathematical computation on the EEs without worrying about compatibility issues. For instance, multiple LV images can be created on demand, each one containing only the code necessary to process the data of a single sensor. The partial results will be then subsequently uploaded to a more powerful edge device (e.g. edge data-center) to be merged. Figure 5 provides more details regarding how unikernels can support both the execution of specific algorithms related to air pollution control and provide pre-processing of input data for simulations running in the Cloud.

The described approach can reduce the load on the core network, end-to-end latency and also the cloud (and air pollution stations) provisioning costs. Regarding specific LV technology, we consider unikernels a promising candidate. The algorithms used to assess air pollution levels are generally static and stateless. In other words, they can be considered as black-boxes with a defined range of inputs/outputs. In case of necessity, the algorithm can be simply changed by replacing it with a new unikernel instance without incurring a long network transfer time[2].

C. Augmented Reality

Wearable devices are typically resource-constrained compared to computer hardware of same vintage PC. The core features of a wearable/mobile device are light-weight, comfort, design and battery life. CPU speed, memory and system capabilities are only secondary, contrary to what are required by the PC market. Therefore, it is not surprising that, overall, wearable/mobile devices are not designed to run computationally intensive tasks.

A common approach to solve the problem is offloading AR tasks to cloud services in order to reduce the power consumption on the device and cope with, eventually, insufficient mobile processing. The drawback is that using cloud service will introduce additional latency, which is crucial for real-time applications. This is especially important for AR applications, where responsiveness and user immersion are paramount. Humans are extremely sensitive to delays affecting real-time interactions (e.g., a phone call). Different studies revealed the speed at which the human brain can identify faces in a dark scene and the requirements of virtual reality application to achieve perpetual stability [13]-[14]. Longer delays in such highly interactive and multimedia-based applications will lower the end-users' experience.

An use cases where there is a strong interplay between local computational resources and AR (or, broadly speaking, computer vision) is augmented windshields for autonomous vehicles. The driver, at this point, passive, might shift is attention completely on the windscreen instead of checking the console in search for speed information. Additionally, the windshield will also provide traffic conditions information, personal agenda, news feed, gaming interfaces, social networks and so forth. In order to craft and manage such a visually-rich experience, an edge board mounted on the car is considered necessary.

Therefore, with the support of edge computing and LV, we have the possibility to offload expensive image processing tasks to EEs in proximity instead of resorting to the cloud back ends. Therefore, we can limit the latency impact, assuming that the computation time is *device-invariant*. The use of virtualization in such context is additionally motivated by the following factors: *multi-tenancy* (i.e., multiple users executing multiple tasks) and *tasks isolation* for privacy.

For this specific use case, a combination of Docker and Unikernel represents a potential approach, as shown in Figure 6. A Docker image containing multiple unikernel can be composed and shipped, each one representing a different AR stage/task. Therefore, the Docker image can offer the orchestration and control API to external applications while, under the hood, unikernels would take care of running the required computations.

VI. Open Issues and Challenges

In this section, we discuss the technical challenges for integrating LV into IoT edge computing and further identify open directions for future research.

A. Orchestration and Monitoring

Orchestration of edge elements (EE) and cloud architectures brings several challenges. Edge-IoT scenarios require specific tools to deal with the different processor architectures and

[1]Usually gas detection sensors (NO, NO_x, O_3, CO, CO_2 and particulate matter) plus humidity, rain detection and wind speed/direction.

[2]Unikernels are, by design, much smaller than other virtualization techniques.

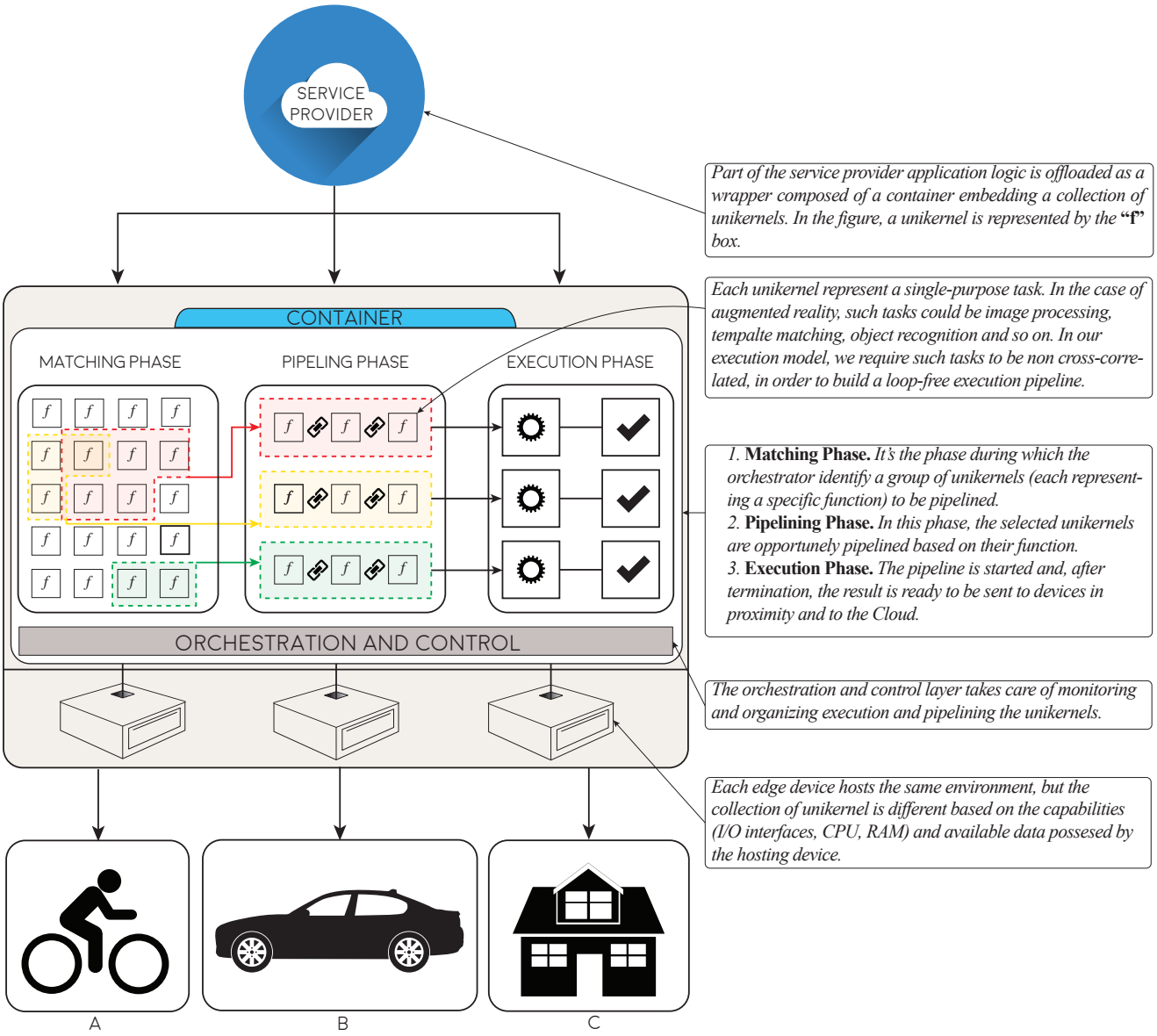


Figure 6: A. Biker receiving personalized advertisements rendered in augmented reality on his smart glasses; B. A smart car populating its augmented windshield with contextualized, live feed information; C. Augmented smart home, where we control IoT devices in proximity through virtual interfaces.

storage capacity of EEs and cloud services. Controlling the network traffic requires to orchestrate cloud and edge, an increasingly challenging task with manifold EEs deployed. Hence, it becomes crucial deploying lightweight orchestration modules that do not overburden the EE, and that seeks a fair balance between synchronization and network load. Other key aspects concern the definition of optimized policies for an efficient *vertical scaling*, in which applications are automatically prioritized and scaled up/down between EE and Cloud, according to specific QoS requirements or computing resources saturation of EEs. Mobility is also a relevant aspect. User devices might move in relation to the edge-processing device providing the service. Therefore, the service may need to be re-deployed multiple times at different locations to serve transparently mobile users. In particular, if the service is specific to an individual user, the number of transfers may be high. Destroying and re-deploying is preferred instead of moving the service together with its running state. For cloud-native service the general recommendation is to avoid storing state locally or only to use disposable state. For services requiring local state, the service must store the current state at an external stable location before exiting and load it again on restart. Particular attention must be paid to ensure that the new edge node has available resources for serving the new device, and the platform may provide alternative nodes or prioritization among services in case of over allocation.

As regards as *monitoring solutions*, both technologies need high-performing, lightweight and scalable monitoring frame-

works. This requirement is strictly related to the fact that these tools may need to run on EEs characterized by lower resource computation capabilities. Another key requirement for monitoring engines is the possibility to track on real-time the individual resources' of each virtualized instance. Implementation of such frameworks becomes, in parallel with orchestration mechanisms, crucial in resource optimization and on developing efficient edge-cloud instances-placement algorithms and policies.

B. Security and Privacy

In the analyzed domain, one challenge is the *certification* of virtualized applications. We need to guarantee their authenticity and validity, by including a signing and validation infrastructure to discriminate *legit* from *tampered* instances. Without such mechanisms, there is the concrete risk of executing malicious code and infringe the security requirements. It is crucial to encourage the development of lightweight security mechanisms, which take into account the strict requirements of IoT applications/scenarios and not impair the lightweight features of the analyzed virtualization technologies, preserving their capacity to not generate performance overhead. From the privacy perspective, EEs may be shared between multiple tenants. It is crucial to be able to isolate tenants' data, but also controlling the use of tenants' dedicated resources — e.g., CPU and memory. Finally, sharing data between tenants at the EE level, without going through the cloud-infrastructure, requires the definition of EE policies and specific access control mechanisms.

C. Standards and Regulations

IoT and EC are developing faster than standards and regulations. The presence of multiple industry partners and researchers working in this field gave birth to different ramifications and interpretations of the same paradigm. Without standards and regulations, merging different approaches will be a non-trivial task exacerbated by the heterogeneity of the involved technologies. For LV technologies, lately there has been a growing effort to lay some guidelines and describe the challenges in the process of building NFV platforms [5]. Nevertheless, this only partially covers the type of functionalities we advocate to offload to EE nodes. Therefore, we consider necessary an additional standardization effort which seeks to lay down precise guidelines towards the employment of LV in a wider range of IoT use-case scenarios.

D. Elasticity in service provisioning

This feature is strictly dependent to the LV engines capacity of quickly allocating/deallocating virtualized instances. Data reported in Table 3a clearly show how both container and unikernel can promptly scale up/down. Furthermore, LV API also allow to *freeze* the execution of an instance and quickly restore it through checkpoint/restore mechanisms. However, there is still a lack of research to evaluate the interactions among multiple EEs, without neglecting that current LV engines implementations not provide fully support for live migration. Specific frameworks that support proactive service migrations for *stateless* applications have been already proposed [15]. However, support for *stateful* applications migration need to be soon integrated for fully exploiting LV benefits in these scenarios.

E. Management Frameworks and Applications portability

Employment of containers technologies have had disruptive rise in the last years, and the enormous effort that open source communities have provided on continually improving fully featured management frameworks has paid off. Unikernels seem to be still not enough mature for being included in production-ready environments, and a greater effort is required for featuring the same *portability* of containers. Packaging applications through unikernel may require an implementation effort that somehow slows down, and in some cases limits, the adaptability towards existing software and hardware platforms. This difference comes from the different way in which the two technologies are built. Containers are application agnostic while unikernels are limited by the programming language and libraries exposed by the underlying minimalistic OS.

F. Data Storage

Containers and, in particular, unikernels are not suitable for storing persistent data, such as data collected from IoT sensors. Moreover, storing important data on edge nodes can be risky both because of the volatile nature of edge nodes and because of the security risks related to easier physical exposure of the nodes. Therefore, data typically needs to be stored in centralized nodes and retrieved on demand. This may reduce the feasibility of LV based edge computation in very data intensive applications. Moreover, some applications requiring nodes to access data of all other nodes data, e.g. for distributed analytics, may be unfeasible to distribute. Automatically optimizing the data storage location of distributed applications is a topic requiring further research. On the other hand, many IoT applications use volatile data locally while persistent data can be minimized and stored centrally.

G. Telco Industry Readiness and Perspectives

The telecommunication sector is currently in a major paradigm shift moving in the direction of *softwarization* of the former hardware based network elements – a concept called NFV [4]. As a first step, the current network functions are directly mapped to corresponding virtualized versions implemented as VMs. The fifth generation (5G) will move toward a more cloud native approach, where different network functions are divided into smaller components that can be individually deployed and scaled and communicate to each other using a message bus. Using MEC as a platform, virtualized network functions (VNFs) can be placed at the edge of the network, and decomposition further encourages the use of LV technologies and the allocation of individual service components to the edge. From the operator perspective, *edge* typically means the base station, but virtualization on Customer Premise Equipment (CPE), such as residential

gateways, may extend the edge further. NFV is the main driver for edge computing in the mobile networks, and a necessity for opening the operator network for third party applications. While operators may have difficulties competing with established players in the cloud market, their presence close to the user make them more competitive for edge-dominated computation. The adoption of LV technologies in telco networks requires a change of mindset in the industry but also technology questions remain for ensuring the reliability and security required for telecommunication networks. As unikernels can be deployed on the same hypervisors as VMs with minor impact on orchestration infrastructure, they are more likely than containers as replacements for VMs.

VII. Outlook

In this article, we examine the challenging problem of integrating LV with IoT edge networks. We first discuss which are the current issues involving EC and IoT network architectures. Therefore, we present three different IoT use-cases, in which LV solutions can bring a set of benefits and a desirable *design flexibility*. Our analysis provides a clear holistic vision of such integration, which promotes innovative network designs to fully exploit the advantages of LV and IoT resources. Finally, we also discuss key technical challenges and identify open questions for future research in this area.

References

[1] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for Smart Cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, 2014.

[2] V. Cozzolino, A. Y. Ding, and J. Ott, "Fades: Fine-grained edge offloading with unikernels," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, ser. HotConNet '17. New York, NY, USA: ACM, 2017, pp. 36–41.

[3] K. et al., "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[4] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, Oct. 2014.

[5] S. Natarajan, R. Krishnan, A. Ghanwani, D. Krishnaswamy, P. Willis, and A. Chaudhary, "An analysis of lightweight virtualization technologies for nfv," 2017. [Online]. Available: https://tools.ietf.org/html/draft-natarajan-nfvrg-containers-for-nfv-03

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[7] R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[8] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 461–472.

[9] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. J. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels." in *NSDI*, 2015, pp. 559–573.

[10] M. Gerla, E.-K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 2014, pp. 241–246.

[11] R. Morabito, R. Petrolo, V. Loscrı, N. Mitton, G. Ruggeri, and A. Molinaro, "Lightweight virtualization as enabling technology for future smart cars," in *International Symposium on Integrated Network Management (IM)*, 2017.

[12] M. Doering, "High-resolution large-scale air pollution monitoring: Approaches and challenges," in *Proceedings of the 3rd ACM International Workshop on MobiArch*, ser. HotPlanet '11. New York, NY, USA: ACM, 2011, pp. 5–10.

[13] S. R. Ellis, K. Mania, B. D. Adelstein, and M. I. Hill, "Generalizeability of latency detection in a variety of virtual environments," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 48, no. 23. SAGE Publications Sage CA: Los Angeles, CA, 2004, pp. 2632–2636.

[14] M. B. Lewis and A. J. Edmonds, "Face detection: Mapping human performance," *Perception*, vol. 32, no. 8, pp. 903–920, 2003.

[15] I. Farris, T. Taleb, H. Flinck, and A. Iera, "Providing ultra-short latency to user-centric 5g applications at the mobile network edge," *Transactions on Emerging Telecommunications Technologies*, 2017.

Publication II

Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. FADES: Fine-Grained Edge Offloading with Unikernels. In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet '17). Association for Computing Machinery, New York, NY, USA, 36–41. DOI:https://doi.org/10.1145/3094405.3094412.

Publication Summary

In recent years, edge computing and Internet of Things (IoT) have become closely coupled. IoT was initially conceived as extending the Internet with a new class of devices and use cases (e.g., personal devices, constrained networks). Early architectures and frameworks introduced the notion of cloud-dependent IoT deployments, with the assumption that most/all IoT edge networks need to be connected to the cloud. We noticed how this tight coupling between cloud and IoT is not desirable in some cases, which we analyzed in our work. For example, when: a) data needs to be processed at the edge, b) delay sensible applications require real-time responses, or c) the amount of data is too large to upload to the cloud (in real-time) without congesting the backhaul.
To address this situations, we created FADES: an edge offloading architecture that empowers us to run compact, single purpose tasks at the edge of the network to support a variety of IoT and cloud services. We designed FADES to efficiently exploit the resources of constrained edge devices through fine-grained computation offloading. We took advantage of MirageOS unikernels to isolate and embed application logic in concise Xen-bootable images. We implemented FADES and evaluated the system performance under various hardware and network conditions. The results show that FADES can effectively strike a balance between running complex applications in the cloud and simple operations at the edge. In our experiments, we revealed the limitation of existing IoT hardware and virtualization platforms, which shed light on future research to bring unikernel into IoT domain.

Author's Contribution

I came up with the idea for the paper as a foundation for an unikernel-based orchestration system. I have designed, implemented, and evaluated the entire system.

FADES: Fine-Grained Edge Offloading with Unikernels

Vittorio Cozzolino
Technical University of Munich
cozzolin@in.tum.de

Aaron Yi Ding
Technical University of Munich
ding@in.tum.de

Jörg Ott
Technical University of Munich
ott@in.tum.de

ABSTRACT

FADES is an edge offloading architecture that empowers us to run compact, single purpose tasks at the edge of the network to support a variety of IoT and cloud services. The design principle behind FADES is to efficiently exploit the resources of constrained edge devices through fine-grained computation offloading. FADES takes advantage of MirageOS unikernels to isolate and embed application logic in concise Xen-bootable images. We have implemented FADES and evaluated the system performance under various hardware and network conditions. Our results show that FADES can effectively strike a balance between running complex applications in the cloud and simple operations at the edge. As a solid step to enable fine-grained edge offloading, our experiments also reveal the limitation of existing IoT hardware and virtualization platforms, which shed light on future research to bring unikernel into IoT domain.

CCS CONCEPTS

• **Information systems** → *Data management systems*; • **Applied computing** → *Service-oriented architectures*; • **Computer systems organization** → *Distributed architectures*;

KEYWORDS

Edge Computing, Virtualization, IoT

ACM Reference format:
Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of HotConNet '17, Los Angeles, CA, USA, August 25, 2017,* 6 pages.
https://doi.org/10.1145/3094405.3094412

1 INTRODUCTION

Edge computing and Internet of Things (IoT) have become closely coupled in recent developments. IoT was initially conceived as extending the Internet with a new class of devices and use cases (e.g., personal devices, constrained networks). Early architectures and frameworks introduced the notion of cloud-dependent IoT deployments, with the assumption that most/all IoT edge networks need to be connected to the cloud. However, there are some cases in which this tight coupling between cloud and IoT is not desirable. For example, when: a) data needs to be processed at the edge, b) delay sensible applications require real-time responses, or c) the amount of data is too large to upload to the cloud (in real-time) without congesting the backhaul. Based on this observation, we advocate a *divide and conquer* approach where IoT and edge devices actively participate in the completion of a task instead of being passively polled by cloud services.

In this context, a key domain that will benefit from the interplay of edge and IoT is Smart Cities: complex environment with manifold IoT devices deployed by different providers and serving many purposes. Moreover, each device would posses different computational and sensory capabilities with varying geographic locations adding to the system convolution. Edge devices will have the responsibility to handle adequately the IoT resources and execute tasks following the back-end instructions. Meanwhile, not everything can be offloaded. A classification based upon application complexity, priority, criticality, power consumption and required physical resources can help in assessing which task to offload. The combination of hardware capabilities and software requirements will ultimately guide the choice.

We define this approach as *edge offloading*. Edge [1] offloading revisits the conventional cloud-based computation offloading, where mobile devices resort to resourceful servers to handle heavy computation [6]. To cater for the demands of IoT services, our approach is reversed: we promote a paradigm where computation is dispatched by the servers to constrained devices deployed at the network edge, close to users and data generators.

To enable edge offloading, recent technological breakthrough in virtualization has provided us new opportunities. For instance, Docker completely revisited the concept of VM by introducing containers. Containers focus on virtualizing at the operating system level, whereas other hypervisor-based solutions focus on abstracting the hardware layer. In the process of creating smaller and more specialized VM, unikernels have emerged as a promising technology. In essence, unikernels are single-purpose appliances that are compile-time specialized into standalone kernels [7]. Unikernels contain exclusively the application code guaranteeing a reduced image size, improved security and greater manageability.

Our main contribution is FADES (*F*unction virtuliz*A*tion bas*ED* *S*ystem), a modular system architecture designed for IoT edge offloading. To achieve reliability, scalability and flexibility, FADES takes advantage of MirageOS unikernels to isolate and embed application logic fragments in small, Xen-bootable images. We decided to adopt the unikernel technology for our implementation because it fits exactly our requirement to run single-purpose tasks without offloading complete application logic. Security is a core benefit but not the main motif behind our choice. Besides system design, the

HotConNet '17, August 25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5058-7/17/08…$15.00
https://doi.org/10.1145/3094405.3094412

[1] Currently, there is no accepted definition of the difference between Edge and Fog. In this paper, we will use the term Edge to refer to groups of constrained devices deployed at the edge of network and able to process local data.

lessons learned from our experiments across different IoT hardware and platforms also shed light on future research to integrate unikernel into the IoT domain.

The rest of the paper is structured as follows: motivation and background (§ 2), related work (§ 3), system architecture description (§ 4), system implementation (§ 5), system evaluation (§ 6), discussion (§ 7) and finally future work and conclusions (§ 8).

2 MOTIVATION AND BACKGROUND

Simplicity is key to the IoT. Regardless of the back-end services, edge devices have to execute simple operations on data locally available. Therefore, by splitting a complex application into manifold simple and single-purpose tasks we can ship them in the shape of lightweight containers (specifically, unikernels). Task fragmentation grants also the possibility to hide the complete application logic for security concern.

IoT diversity and cardinality are like double-edged blades. On one hand we face the problem of heterogeneity and lack of standards. On the other hand, the massive scale of envisioned devices is a powerful source to harness. The core motive behind our system is exactly to leverage this power in the right way. The combination of locally available resources, computational power and capabilities are key elements to properly offload tasks and, therefore, exploit IoT resources. Moreover, heterogeneity is less troublesome when we have intermediate nodes supervising and managing clusters of IoT devices instead of entrusting all this knowledge and responsibility to the cloud. This multi-level (cloud, edge and IoT) information pipeline is also motivated by the necessity of reducing the uplink access parallelism. By offloading computation we progressively aggregate data along the pipeline reducing massively the data to be uploaded and easing the burden on the cloud. Latency is also affected when we move computation closer to the data to be manipulated. However, not everything can be offloaded. Parameters as application complexity, priority, criticality, power consumption and required physical resources have to be taken into account.

Last but not least, security and privacy in IoT are increasingly important [12]. Small IoT devices are not powerful enough to guarantee the required degree of security. Therefore, we advocate the presence of intermediate control units across the communication pipeline between IoT and Cloud in a way to introduce additional resiliency and control. Privacy enabling modules can be deployed directly next to the source of the data, combined with security controls. Still, these features wouldn't be possible without a hardened execution environment. Our design principles offer functionality isolation and reduced attack surface. Moreover, deployed tasks run inside a virtualization platform (hypervisor) adding an additional layer of isolation and protection (subverting the system requires to find vulnerabilities in the hypervisor rather than the OS, which is more difficult).

3 RELATED WORK

Our work follows the trend of exploiting computational resources outside cloud deployment combined with use of unikernels [2, 7, 14]. Computation and data offloading has been explored in different flavors in [4, 5, 13] for aspects as energy efficiency and offloading decision policies. Instead, [15] is a pioneer in the field of computation offloading supported by VMs migration. Their approach is based on leveraging infrastructure resources based on mobile nodes proximity with the support of cloudlets: a trusted, resource-rich computer or cluster of computers. To this end, they adopt dynamic VM synthesis: a process of merging a static part of a VM with a dynamic component provided by a mobile device.

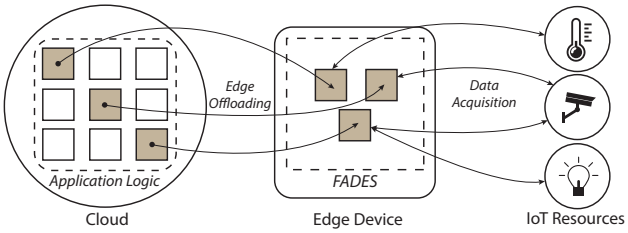


Figure 1: Edge Offloading

Unikernels have been explored in multiple research fields. Some research directions that make use of unikernels to improve existing services or propose new system architectures are [1], [10], [11], and [17].

More recently, Madhavapeddy et al. [8] proposed on-demand specialized VM instantiation. We leverage the idea of the *embedded cloud* presented in this paper and bring it further into the IoT domain. Compared with Jitsu, our work is geared towards IoT services and focuses on the system architecture. Additionally, our evaluation revealed the limits of unikernels by studying the performance with bigger payloads and in different network settings.

Airbox [3] presents a software platform based on onloading and backend-driven cyberforaging. It shares the general direction presented in our paper in terms of offloading the Edge Functions (EF). Compared with their solution, FADES achieves fine-grained offloading by using unikernels instead of Docker technology.

Databox [9] proposes a hybrid physical and cloud-hosted system for personal data management. The prototype of Databox is based on Docker. The authors have indicated MirageOS as a possible candidate to implement future extensions. To this end, FADES is a unikernel-based system dedicated for IoT edge offloading. Inspired by one of the use-cases of Databox, we further share our insights of processing sensors data at the edge through unikernels.

4 SYSTEM DESIGN

Data locality is the key property that drives our system design principles. We see into this physical distance between cloud and IoT a gap to be exploited. Therefore, our design introduces an intermediate unit to enrich and augment the interaction between IoT resources and External Services (ES), as depicted in Figure 2.

The IoT resources offer physical capabilities to interact with the environment and carry out basic tasks. The ES are back-end applications interacting with our system by offloading parts of their operation logic. In our design, we consider ES as a repository of deployment-ready tasks designed for different scenarios and purposes.

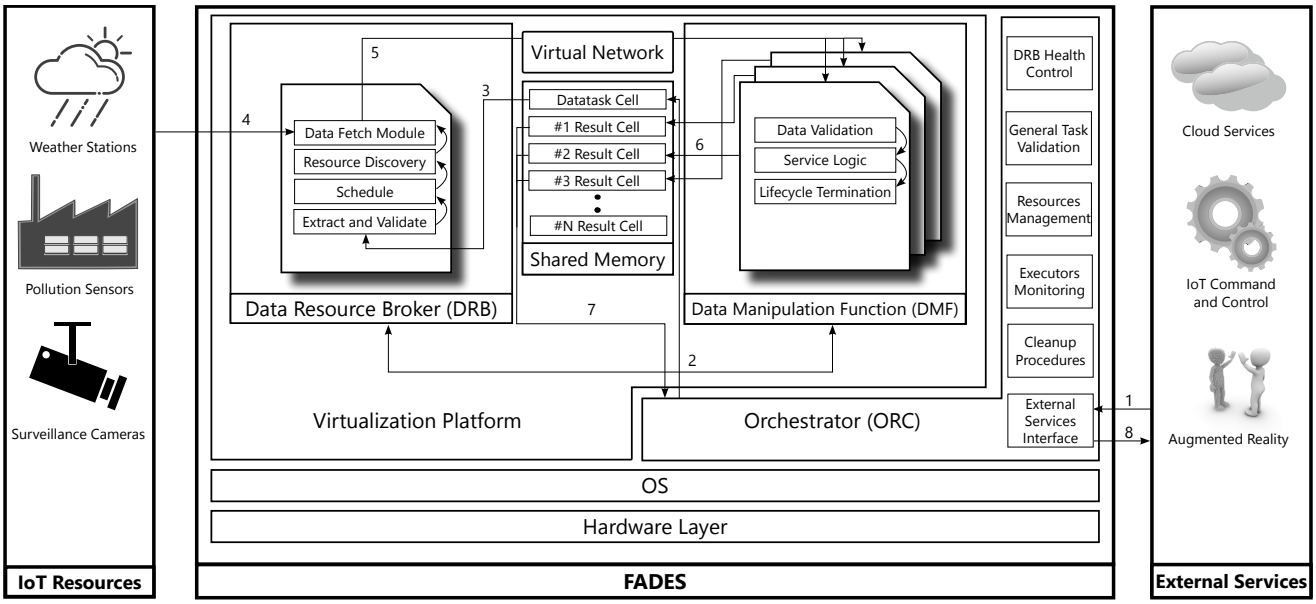


Figure 2: System Design

Recalling the Smart City example, pollution control is achieved by querying pollution sensors in the IoT, aggregating the information at the edge, and sending the final result to the cloud. In this regard, FADES resembles a middlebox and oversees groups of IoT devices based on spatial proximity, as Figure 1 shows. Therefore, each FADES unit supervises a subset of IoT devices (e.g. pollution sensors) scattered across the Smart City. Moreover, it has the responsibility to map and monitor the available IoT resources and retrieve data from them following the offloaded task requirements.

Based on the definition of edge offloading, our system is designed to support only pull workflows (from the cloud to the IoT). Push workflows, where edge devices offload tasks to the cloud, are not yet supported mostly because covered already in mobile offloading research.

The main components of FADES include Orchestrator (ORC), Data Resource Broker (DRB) and Data Manipulation Functions (DMF). As shown in Figure 2, FADES is an event-based system that responds to external commands referred to as *Metadata Task Wrapper* (MTW), which are issued on-demand by services and applications. The MTW can be divided into 3 types:

MTW Credentials: This type contains passport-like information (e.g., task ID, associated user or service, priority). It's mainly used to keep track of the received MTW and schedule its execution.

DRB Metadata: This type contains a list of *data retrieval operations* (DTO), which conveys details about what data to retrieve. DRB Metadata also specifies the source and destination of the data.

DMF Metadata: This type contains application specific details about the DMF. For instance, the MTW issuer can specify additional information about extra runtime configuration parameters or minimal required hardware resources.

4.1 FADES

The core of FADES consists of three components:

Orchestrator (ORC). The Orchestrator is the interface between the system and the outside world. Being a "supervisor" that monitors and controls the system, ORC frequently checks that both DRB and DMF are running correctly. ORC also takes care of dispatching the required information to the DRB the moment a new MTW arrives. It ensures the overall system integrity by monitoring the DRB, following the life-cycle of each DMF, validating uploaded tasks and decommissioning terminated DMFs.

Data Resource Broker (DRB). The DRB module localizes and extracts resources from a groups of IoT devices. Hence, it possesses the required knowledge regarding which IoT devices to query. The DRB is completely computation agnostic, whose execution cycle is event-based and driven by DRB Metadata contained into the MTW and received by the orchestrator.

Data Manipulation Functions (DMF). A DMF embeds exclusively the relative service logic and stays in a dormant state until it receives the correct data from the DRB. DMFs can be persistent or ephemeral, depending on the embedded application logic. Long running or recurrent tasks might exhibit a persistent behavior while single-execution tasks have to be deallocated after completion.

Recalling the pollution control example, we can map each FADES component onto the following roles: 1) the ORC receives from cloud the tasks to be executed and additional metadata, 2) the DRB will locate the correct pollution sensors (or weather stations) to be queried and retrieve the data, 3) the DMF will manipulate the pollution data received from the DRB, produce the final results and send them to the ORC.

5 IMPLEMENTATION

FADES is a virtualized, unikernel-based system hosted by the Xen hypervisor (except for ORC). Our tool of choice is the MirageOS library operating system, which is specifically designed to build modular systems and runs natively on Xen. MirageOS offers static type-safety combined with single-address space layout. Moreover, being compile-time defined and sealed, any code not presented inside a MirageOS unikernel during compiling time will never be executed, and hence preventing any sort of code injection attack [7].

In FADES, both the DRB and DMFs are MirageOS unikernels running on Xen as Para-Virtualized Machines (PVM). The former exclusively retrieves information and the latter process the received data. We enforce component isolation and independent development by confining functionality overlapping.

The DRB is implemented as a daemon unikernel. For internal communications, it uses two modules offered by Xen: the virtual network and the XenStore. The virtual network is used to internally transfer data to the DMFs. On the other hand, XenStore is exclusively used to exchange synchronization messages with the ORC. The DRB validates and schedules each MTW received from the ORC. In our implementation, we implemented different scheduling policies (e.g. sorting by task priority, execution deadline, task ID) but in the current stage we used a simple FIFO. Currently, the DRB is able to process in parallel multiple *data retrieval operations* but is only able to prosecute a single MTW at a time.

The DMFs used in our implementation execute simple aggregation operations over streams of sensors data (Section 6 will cover more details about this choice). The result of the computation is sent to the ORC through Xenstore. DMFs are hooked on the same virtual network of the DRB and do not have external network access. One limitation of using the virtual network is that we need to manage carefully the IP addresses to avoid collisions. In our current implementation, we didn't implement any automated deployment functionality covering this matter. The system currently supports parallel execution of multiple DMFs even thought it has not been fully tested on that regard.

The ORC is developed in Python and it's the only non-virtualized module in FADES. We choose Python because it has the right combination of performance and features that make prototyping fast and flexible. Moreover, the ORC is the only module that handles reads and writes towards the persistent storage. Our design choice focused on establishing a loose coupling between the host system and the unikernels managed by FADES, with the latter being exclusively dependent on *virtual* resources: CPU, RAM and network. In order to be independent from the hosting edge device, the DRB and DMF should be as flexible as possible and ephemeral.

6 EVALUATION

The goal of our evaluation is answering the following questions:

Q1. How different architectures (x86, ARM) affect the performance of MirageOS? What are the Unikernel PVM memory sizing requirements in relation to the amount of data to be manipulated?

Q2. How does our system perform under different workloads and what is the overhead introduced by using multiple modules?

Q3. How much edge deployed services can benefit from data locality? What is the trade-off?

For our tests, we selected three different devices: a Cubietruck (Allwinner A20 ARM Cortex-A7 dual-core @ 1GHz, 1GB RAM, 100Mb Ethernet), an Intel NUC (Intel(R) Core(TM) i5-6260U CPU @ 1.80GHz, 16GB RAM, 1000Mb Ethernet) and a Dell PowerEdge R520 (Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz, 140GB RAM, 1000Mb Ethernet) running Xen 4.4, InfluxDB v1.0 and an Ubuntu 14.04 dom0. Additionally, we used MirageOS 2.9.1 (the latest stable version at the time of writing), OCaml 4.02.3, OPAM 1.2.2 and Python 2.7.6.

We gleaned the data for the tests from our Intel Edison IoT testbed. The testbed is composed by 5 Intel Edison boards deployed in different office rooms on the campus. Each board continuously collects environmental data through a set of sensors (humidity, temperature, light intensity, audio, proximity). At the time of writing, the testbed collected roughly 300 millions of data points equally divided among the 5 sensor classes. Each data point is a row in our database containing: timestamp, sensor value, sensor type, measurement unit and location. The deployment has been running constantly since June 2016.

The testbed is to carry out user-context modeling and correlate sensors readings with human behaviors/actions in each room. Some of the aggregation operations carried out on the testbed have inspired the algorithm embedded in our unikernels. Therefore, the DMFs used for our test execute simple manipulation and aggregation functions (e.g., calculate minimum, maximum, average) over sensors data streams.

Memory Analysis. When offloading to resource constrained devices, it's important not to abuse the already limited available hardware resources. In this section we analyze how the available RAM memory affects DMFs performance. This study serves as a guideline to the process of correctly dimensioning a MirageOS PVM. Therefore, we aim at avoiding the over/under dimensioning issues that could lead to, respectively, waste of resources and out of memory exceptions.

Figure 3 shows the ratio between available heap memory and pre-allocated RAM with different architectures (x86 and ARM). On both architectures the effective available memory is lesser than the amount allocated at the beginning but the behavior varies between x86 and ARM. In the first case, the gap is much more prominent, and this directly affects the amount of data processable by the Unikernel especially in the case of PVM with low allocated RAM memory.

Two main factors influence the available memory for an unikernel PVM: underlying architecture and imported libraries. We cannot economize on the latter, given that it depend on the specific application logic. On the other hand, different system architectures constantly generate different unikernel images. While on ARM the building output is a Linux kernel ARM boot executable zImage (.xen) plus a ELF 32-bit LSB executable (.elf), on x86 we have a single .elf file. The ultimate difference in size of the generated Xen image on the two architectures affects directly the available memory at runtime.

Figure 4 shows the correlation between pre-allocated RAM and maximum amount of processable data. These data has been obtained by studying in details the performance of a single DMF completely isolated from the system. We monitored its limits by feeding an

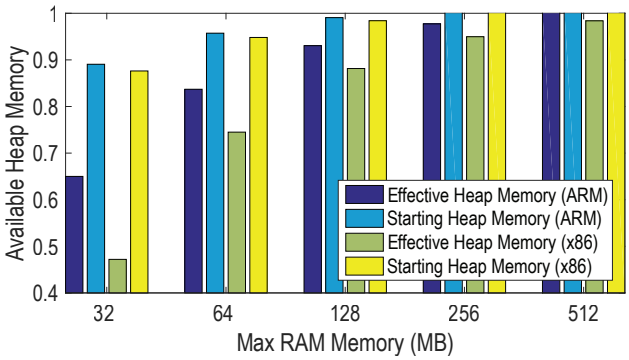


Figure 3: Memory Analysis

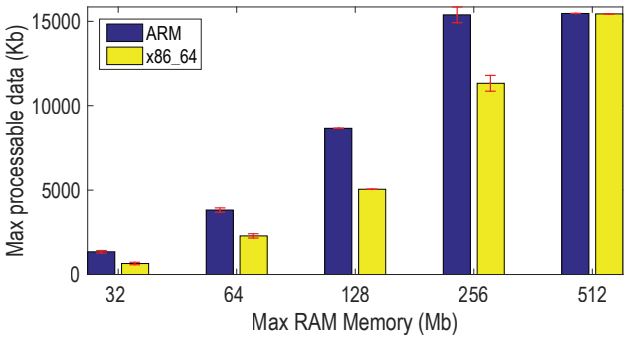


Figure 4: Data processing limits (memory)

increasing amount of data to process. From our tests, we noticed that the device resources doesn't influence at all the maximum amount of processable data. Hence, the graph shows a generic comparison between ARM and x86.

The gap between these two architecture is much more prominent in the case of PVM equipped with low RAM memory, whereas it tends to disappear with 512MB. In some cases, we are only able to process half of the data on x86 architecture. The red error bar is the standard deviation.

System Analysis: Overhead and Offloading. The purpose of this section is to show a performance comparison when executing task at edge instead of in the cloud.

Figure 5 shows an answer to **Q2** by highlighting the system performance with different data payload sizes and providing a detailed breakdown of the execution time of a task in FADES. Four main factors affect the overall execution time: 1) required time to boot-up the DMF unikernels, 2) time required to transfer the data between the DRB and the DMF, 3) computation time of the DMF and 4) the time required to retrieve the data to manipulate.

In particular, factor 1) is an aggregated value representing the overhead introduced by the ORC module. For now, the ORC is a thin layer that handles requests from the external services and doesn't process data or applies any changes to the tasks executed by the DMFs. It checks that everything works correctly but doesn't take part in any computation phases. Factor 4) only affects the scenario where the Dell PowerEdge server has to retrieve data from

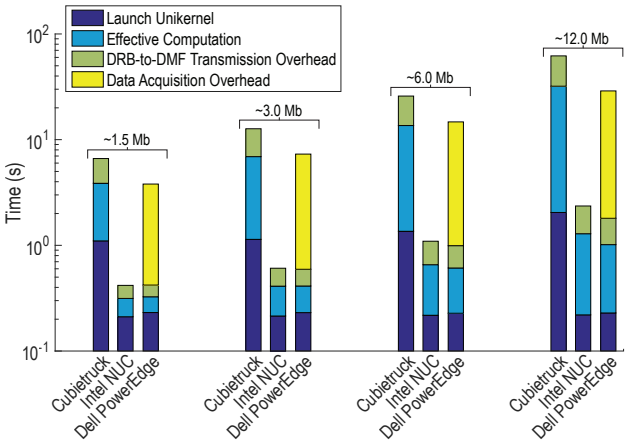


Figure 5: System Performance

a remote network. The bars are grouped by amount of data to be processed. For example, the first group shows the performance for each devices given a payload of 1.5Mb.

The results show that the presence of a sufficiently powerful device at the edge of the network combined with data locality makes edge offloading the best decision. Particularly, the Intel NUC outperforms the Dell PowerEdge while the Cubietruck is highly hindered by the intra-unikernel transmission time overhead.

Figure 6 shows an answer to **Q3** by presenting in details how data locality strongly affects performance regardless of the hardware capabilities. Therefore, we focused on observing how data locality affects computation time. The Cubietruck and the Intel NUC had a local copy of the sensors data while the Dell PowerEdge (the cloud) was forced to retrieve the data from a remote location. To this avail, we deployed our Intel NUC and the Cubietruck into another building and used them as a data source. In the graph we can clearly see that having data locally can improve performance even on resource constrained devices.

The remote data location is accessed by the Dell PowerEdge through a standard broadband connection with the following specifications: 45.38 Mbps downlink and 0.574 Mbps uplink (effective).

Connection speed is a critical factor in our evaluation; we are aware that faster/slower connections will lead to different result. Still, we want to point out that even by reducing to zero the data acquisition overhead factor, the Intel NUC performance are surely vying with the Dell PowerEdge.

The cost of uploading the unikernel has been removed from our evaluation. This factor it's directly bound to the nature of the application logic. In a real scenario, it is very well possible that a DMF could actually be reused multiple times (and by multiple users) before it's updated. In other words, the frequency at which the data changes is greater than the one of the deployed task. On the other hand, the task might need to be updated constantly rendering it obsolete at every new compute cycle. The chances for the latter to happen are dim, yet possible.

Last but not least, we are aware of the asymmetric nature of common broadband subscriptions with a ratio of 1:10 between upload/download speed. Hence, the cost of uploading data from

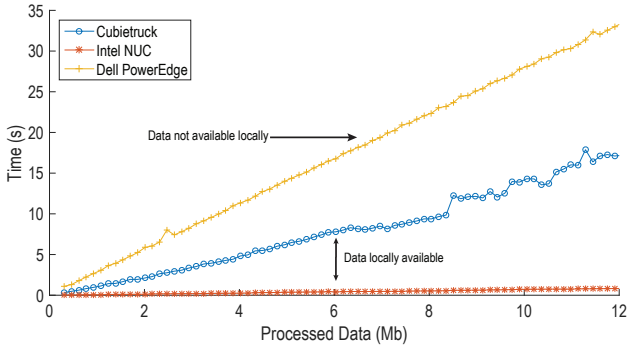


Figure 6: Effect of Data Locality on Computation Time

the Cloud to the edge will generally be marginal compared to the opposite.

7 DISCUSSION

We have learned several practical lessons from system development and experiments, which can be summarized as *hardware limitations*, *platform support*, and *security concern*. First of all, it's demanding to find suitable embedded boards that can support Xen and MirageOS, regardless of x86 and ARM architectures. In most cases, the main culprits are hardware incompatibilities and poor documentation (if there is any). Our experiments on Cubietruck have benefited from the MirageOS discussion group, where we found detailed setup guidelines. On the other hand, the deployment on mini-PC (e.g. Intel NUC) is much easier, since those devices essentially resemble the standard PC.

Secondly, our system performance is affected by the limitation of MirageOS. In particular, we struggled with the network API when transferring data between two unikernels. The main issue comes from a bug in the TCP/IP MirageOS stack that doesn't handle properly writing packets larger than the MTU. In consequence, we had to introduce an extra chunking function at the application layer to split, and later reconstruct the data. This negative effect is reflected in Figure 5 where the overhead of this operation prolongs the completion time, especially on constrained devices like the Cubietruck.

Finally, it takes extra steps to enhance system security. FADES enables an execution paradigm where single-purpose functionalities are offloaded from the service provider (e.g., cloud) to edge devices. We hence need to guarantee the authenticity and validity of the offloaded tasks. Without a signing and validation infrastructure to discriminate legit from tampered unikernels, we might risk executing malicious code and infringe the security requirements. Furthermore, if a FADES module is compromised or hijacked, regardless of the offloaded code, the attacker is able to manipulate the results of trustworthy DMFs. Therefore, a strict control over the system execution pipeline and constant monitoring of FADES is mandatory, as suggested in [16].

8 CONCLUSION AND FUTURE WORK

FADES is a modular offloading architecture that leverages lightweight virtualization to enable fine-grained edge offloading for IoT. The underlying idea is to bridge the gap between complex applications running in the Cloud and simple operations running at the edge. It's in this gap that we spot the opportunity to utilize unikernels as an ideal vessel to ship single-purpose tasks for achieving modularity, flexibility and multi-tenancy. As a first step to explore the potential of lightweight virtualization in IoT and edge computing, our experimental insights shed light on the hardware deployment and performance optimization for both system engineers and researchers.

Our future work will focus on three aspects. Firstly, we plan to investigate the system scalability by running multiple function instances in parallel. Secondly, we will evaluate FADES against real-world applications. Lastly, we will harden the system security design to meet the security requirements of IoT. In addition, we will also compare our system with other existing solutions.

ACKNOWLEDGEMENT

This research was partially supported by Google Internet of Things (IoT) Technology Research Award Pilot. Moreover, it is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) through the Center Digitization Bavaria, an initiative of the Bavarian State Government.

We thank our shepherd, Ryan Huang for the valuable feedback and support. Moreover, we thank also our colleagues from UCI who provided insight and expertise that greatly assisted the research.

REFERENCES

[1] Bob Duncan, Andreas Happe, and Alfred Bratterud. 2016. Enterprise IoT Security and Scalability: How Unikernels Can Improve the Status Quo. In *Proceedings of ACM UCC '16*.
[2] Bratterud et al. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of IEEE CloudCom '15*. IEEE.
[3] Bhardwaj et al. 2016. Fast, scalable and secure onloading of edge functions using AirBox. In *IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE.
[4] Ding et al. 2013. Enabling energy-aware collaborative mobile data offloading for smartphones. In *Proceedings of IEEE SECON '13*.
[5] Kosta et al. 2012. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings IEEE INFOCOM '12*.
[6] Kumar et al. 2013. A survey of computation offloading for mobile systems. *Mobile Networks and Applications* 18, 1 (2013), 129–140.
[7] Madhavapeddy et al. 2013. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, Vol. 48. ACM.
[8] Madhavapeddy et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of NSDI '15*.
[9] Mortier et al. 2016. Personal Data Management with the Databox: What's Inside the Box?. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*.
[10] Sathiaseelan et al. 2015. SCANDEX: Service Centric Networking for Challenged Decentralised Networks. In *Proceedings of ACM DIYNetworking '15*.
[11] Siracusano et al. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of ACM HotMiddlebox '16*.
[12] Jorge Granjal, Edmundo Monteiro, and Jorge Sá Silva. 2015. Security for the Internet of Things: a survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials* 17, 3 (2015), 1294–1312.
[13] Esa Hyytiä, Thrasyvoulos Spyropoulos, and Jörg Ott. 2015. Offload (only) the right jobs: Robust offloading using the markov decision processes. In *Proceedings of IEEE WoWMoM '15*.
[14] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v—Optimizing the Operating System for Virtual Machines. In *Proceedings of USENIX ATC'14*.
[15] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing* 8, 4 (2009).
[16] Rolf H Weber. 2010. Internet of Things–New security and privacy challenges. *Computer Law & Security Review* 26, 1 (2010), 23–30.
[17] Dan Williams and Ricardo Koller. 2016. Unikernel monitors: extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

Publication III

Vittorio Cozzolino, Oliver Flum, Aaron Yi Ding, and Jörg Ott. 2020. MirageManager: enabling stateful migration for unikernels. In Proceedings of the Workshop on Cloud Continuum Services for Smart IoT Systems (CCIoT '20). Association for Computing Machinery, New York, NY, USA, 13–19. DOI:https://doi.org/10.1145/3417310.3431400.

Publication Summary

In edge computing environments, offloading computation to the nearest edge node is key to cutting network latency and improving user experience. We found edge and IoT networks to be unreliable and composed of resource-constrained devices that are more prone to failures. Ideally, the edge infrastructure should be able to self-adapt in case of malfunctions and quickly move the computation to a stable node in order to maintain high service responsiveness and avoid data loss. We identified fast service migration and recovery (e.g. reinstantiation) to play a crucial role in reducing the overall service downtime.

While migration based on VMs or containers in the edge computing domain has been explored in numerous studies, the same is not true for unikernels. In fact, unikernels have only recently gained popularity as an alternative to virtual machines and containers. However, the unikernels ecosystem is still in its infancy and lacks quintessential functionalities found in more well-established virtualization technologies. We identified stateful migration as an a highly desired feature for mobile edge services in distributed environments which is not yet supported by unikernels. Therefore, our work in this manuscript is focused on addressing this shortcoming.

We built MirageManager: a ready-to-deploy unikernel migration system enabling lossless migration supported by a function-level, application logic checkpointing library of our design. Our evaluation results show that MirageManager is able to lower the service downtime by 80%, and drastically reduce the state transfer data by almost 100% when comparing against Podman. Additionally, MirageManager also beats Podman, a container-based engine, in parallel service migration across constrained edge networks reducing the overall migration time by up to 6x.

Author's Contribution

I came up with the idea for the paper as a foundation for the stateful migration of unikernels. I contributed to the system design aspects and research foundation, while Oliver Flum implemented and evaluated the system.

MirageManager: Enabling Stateful Migration for Unikernels

Vittorio Cozzolino
Oliver Flum
vittorio.cozzolino@in.tum.de
oliverflum@gmail.com
Technische Universität München
Munich, Germany

Aaron Yi Ding
Delft University of Technology
Delft, Netherland
Aaron.Ding@tudelft.nl

Jörg Ott
Technische Universität München
Munich, Germany
ott@in.tum.de

ABSTRACT

Unikernels are a new lightweight virtualization technology born as an alternative to virtual machines and containers. Geared towards service provisioning for the Internet of Things (IoT) and edge computing, they offer extremely small memory footprint and strong isolation properties. However, the unikernels ecosystem is still in its infancy and lacks quintessential functionalities found in more well-established virtualization technologies. For example, stateful migration is a highly desired feature for mobile edge services in distributed environments which is not yet supported by unikernels. This is one of the shortcomings preventing us from reaping the full benefits of unikernels outside of stateless applications.

In this work, we aim bridging this gap with MirageManager: a ready-to-deploy unikernel migration system enabling lossless migration supported by a function-level, application logic checkpointing library of our design. Our evaluation results show that MirageManager is able to lower the service downtime by ~**80%**, and drastically reduce the state transfer data by almost ~**100%** when comparing against Podman. Additionally, MirageManager also beats Podman, a container-based engine, in parallel service migration across constrained edge networks reducing the overall migration time by up to ~**6x**.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**;
• **Software and its engineering** → *Software architectures.*

KEYWORDS

Edge computing, service migration, unikernel

ACM Reference Format:
Vittorio Cozzolino, Oliver Flum, Aaron Yi Ding, and Jörg Ott. 2020. MirageManager: Enabling Stateful Migration for Unikernels. In *Cloud Continuum Services for Smart IoT Systems (CCIoT '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3417310.3431400

CCIoT '20, November 16–19, 2020, Virtual Event, Japan

ACM ISBN 978-1-4503-8131-4/20/11…$15.00
https://doi.org/10.1145/3417310.3431400

1 INTRODUCTION

In edge computing environments, offloading computation to the nearest edge node is key to cutting network latency and improving user experience [1–5]. However, edge and IoT networks may be unreliable [6] and composed of resource-constrained devices that are more prone to failures. Ideally, the edge infrastructure should be able to self-adapt in case of malfunctions and quickly move the computation to a stable node in order to maintain high service responsiveness and avoid data loss. Therefore, fast service migration and recovery (e.g. reinstantiation) play a crucial role in reducing the overall service downtime.

Migration based on VMs or containers in the edge computing domain has been explored in numerous studies. VM handoff [7] has been proposed to accelerate service handoff across offloading edge nodes. It divided VM images into two stacked overlays based on Virtual Machine (VM) synthesis [8] techniques. In contrast, the wide deployment of containers platforms provides a base for high speed service handover. The Docker storage driver employs layered images inside containers, enabling fast packaging and shipping of any application as a container. Many container platforms, such as OpenVZ [9], LXC [10], and Docker [11], either completely or partially support container migration, but none of them are suitable for the edge computing environment [5]. In fact, LXC migration and Docker migration are based on Checkpoint/Restore In Userspace (CRIU) [12] and need to transfer the whole container file system during the migration, resulting in inefficiency and increasing network overhead as a function of the filesystem size.

To cope with future Internet architectural trends fueled by a pressing need to support edge/fog computing environments, new forms of service decomposition such as lambda functions [13] and unikernels [14] have been developed. While both approaches are designed for stateless applications, we argue that for specific use-cases it is necessary to preserve the execution state. For example, in security applications that collect network traffic data and trigger events if suspicious behavior is detected [15], this data would be lost in a simple image duplication of stateless migration. Time variant computations, such as sensor fusion, would deliver different results depending on when they start or how long they run. In addition, services deployed in radio access networks that are tightly coupled with their mobile user would have to preserve their data in order to stay in sync with their users physically moving from one access point to another [16]. However, unikernel migration has not yet been explored in past research. Most approaches opted for fine-tuned container-based solutions to reduce service downtime at the edge. Meanwhile, we found no tool offering stateful migration for

unikernels which, due to their properties (discussed in §2), are a promising option for edge computing applications.

To fill this gap, we developed **MirageManager**: a checkpoint-based, live migration solution for unikernels. Our contribution is two-fold: (i) an architecture and workflow to manage the migration of unikernels and (ii) a library called *Unimem* which can preserve the unikernel state at function level during the migration. Our prototype is based on MirageOS and we strive to abstract from any platform-specific implementation details by handling the execution state explicitly within the application itself before transferring it. Initially geared towards MirageOS, **MirageManager** is designed to be later ported to other unikernel implementations.

The remainder of this paper is structured as follows. We provide background information and review related work in §2. We describe our system design and *checkpointing* technique in §3 and §4, respectively. We discuss the details of our implementation in §5 followed by preliminary results in §6. Finally, we discuss pros and cons of our approach in §7 and conclude the paper in §8.

2 MOTIVATION AND RELATED WORK

We begin by looking at the need for stateful unikernel migration in respect to current trends in the edge computing and IoT domains.

With the rapid development of the edge computing model, many researchers have developed applications exploiting the edge computing paradigm. Machine learning, computer vision and signal processing are just examples of classes of applications that benefit from offloading intensive computation [17–19] to nearby edge devices. Similarly, research exploring service migration for edge computing followed [16, 20–22], highlighting pros and cons of the different approaches adopted. Originally, the idea of shipping and migrating computation was supported by code slicing or VM and container migration [23, 24]. However, these mechanisms require transmission of a considerable amount of data over the network due to sheer size of the execution environment that needs to be migrated, which also has energy consumption impact [25]. The advantage is portability since less assumptions need to be made regarding the destination machine.

Containers require less data to be transferred as the OS is not migrated during the procedure and are overall more lightweight than VMs [26]. On the other hand, containers depend on OS-specific functions which makes the migration procedure difficult due to the presence of external dependencies [27]. A sweet-spot between these two approaches are *unikernels*: a new, emerging multi-purpose virtualization technology tailored for resource-constrained devices commonly found at the edge [28–30]. Due to their strong isolation properties, reduced attack surface, and small memory footprint, unikernels represent an intersection between containers and traditional VMs. Additionally, their compilation model enables whole-system optimization across device drivers and application logic. In particular, by being in the order of a few MBs in terms of image size, the migration of an unikernel is more resilient to the unfavorable network conditions (e.g., unreliable, intermittent connectivity, limited bandwidth) often found in edge and IoT networks [6]. For example, operations like image or state retransmission are less costly in terms of transmitted data when compared to other virtualization techniques.

Unikernels are still in their infancy and do not offer the suite of functionalities provided by other well-established virtualization tools. This applies also to migration, which, to the best of our knowledge, is still an unexplored path. In fact, unikernels are advertised primarily as stateless appliances which, by definition, do not require migration as no information should be preserved across subsequent executions. However, unikernels have been recently used also in stateful contexts as in [15, 31] where preserving the state of execution is necessary to maintain consistency and avoid gaps in the collected data. Security applications, for example, collect data on network traffic and trigger events if suspicious behavior is detected. For example, in [15] a unikernel based intrusion detection system (IDS) is proposed. In this case, loss of the data structures during service re-instantiation could expose the network to attacks as malicious connection could not be tracked anymore. Additionally, time-variant computations such as sensor fusion will deliver different results depending on when they start or how long they run. Another possible application is motion patterns detection in a video stream, which requires to preserve information from past processed video frames during a migration [32]. However, in this case the unikernel would eventually require access to additional hardware resources (e.g., GPU) for which support is yet not available. For the use-cases mentioned above, stateful migration can help in increasing the service reliability in case of faults and prevent data losses in case of service reinstatiation.

3 SYSTEM DESIGN

Typically, service migration is achieved by dumping the content of a virtual instance into a file and transferring it to a destination host. Then, the hypervisor will take care of restoring the service. However, this procedure also requires support by the guest operating system. While most hypervisors support migration, this is not necessarily the case for all guest OSes. In fact, unikernels do not support it for the reasons explained in §2.

We added the required migration logic directly at the application layer instead of making any changes at the kernel level (e.g., MiniOS [33]) or in the hypervisor. This is a practice followed also in past work for VM-independent migration of stateful applications or to capture the application state at a high-level before migrating it [32, 34]. Hence, we designed a set of functionality in the shape of a library allowing the unikernel to keep track of its own state internally. When the unikernel needs to suspend, it serializes its state so it can be transferred to the migration target, which will process the state before proceeding with the execution flow.

Aside from state tracking (discussed in §4), we require an additional component to support the migration process which we call MirageManager, shown in Figure 1. It is a web service exposing an interface to commission and manage unikernels on any registered host and transfer the unikernel state using a repository. It is the core of our system and it manages the life-cycle (e.g., creation, migration, destruction) of unikernels deployed on multiple hosts and it provides a repository for writing and retrieving the unikernels state before and after a migration. MirageManager is installed at the edge and we assume that there will be multiple instances of it to manage clusters of edge nodes. The representation of a unikernel is populated before the hypervisor creates the guest domain and

will exist even after its destruction, regardless of whether it is the result of a migration procedure or a regular shutdown. During the guest lifetime, the representation will change to reflect changes in its state.

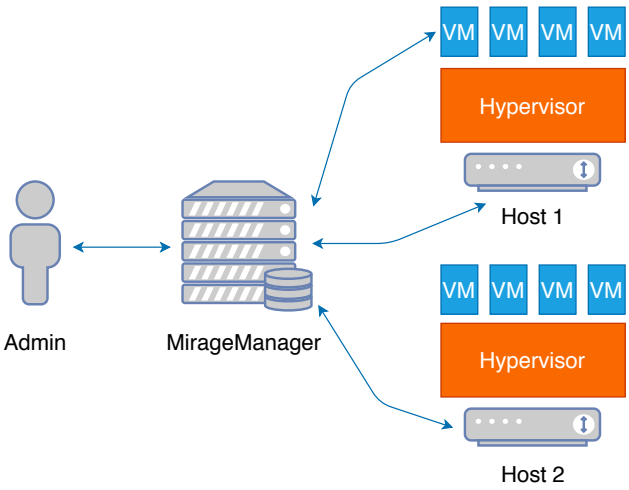


Figure 1: MirageManager.

When a unikernel is started, MirageManager will create a guest domain of the corresponding image on the target host. Afterward, to confirm a successful boot, the unikernel will query MirageManager and start a lookup procedure for a previous state associated to it. This procedure is required so that, even if no state is retrieved, MirageManager will be aware of the current state of the unikernel (specifically, *started*) and change it to *connected*.

At the moment of a migration, MirageManager will issue a *suspend* command to the unikernel. Hence, the latter will transfer its state to the repository and thereby confirm that the suspension was successful. When resumed on the target machine, a state will be retrieved from MirageManager and the guest will use it to update itself before resuming its workflow exactly from where it was interrupted before the migration. This process can be repeated indefinitely until the unikernel is permanently stopped, completes its intended task, or exits due to an error/fault. Potentially, the unikernel could invoke by itself the migration procedure without the need for an external trigger. However, this functionality is not yet supported in the current version of our system.

4 STATE CHECKPOINTING

In order to manage the lifecycle of a unikernel, MirageManager requires a complete representation of its execution state. Therefore, we developed a module able to store and serialize the unikernel application logic state so that execution can be resumed from it. We call this procedure *checkpointing*, described below.

For the purpose of creating checkpoints, we implement a library for MirageOS that defines a central state object representing the unikernel's state. Additionally, we defined a programming model which allows to express the application logic routines in a serializable format, so that the execution state can be written to the store and transferred to the repository. Such store is called *Unimem* and it is implemented as a key-value store using strings as keys and a polymorphic data-type for the values. The currently supported data-types are single element, list, or list of lists. The application can decide to write either variables (e.g., intermediate execution results) or details about its execution state into Unimem, depending on the context. Additionally, Unimem also encapsulates the communication protocol semantics used to dialogue with MirageManager, which are shown in Figure 5.

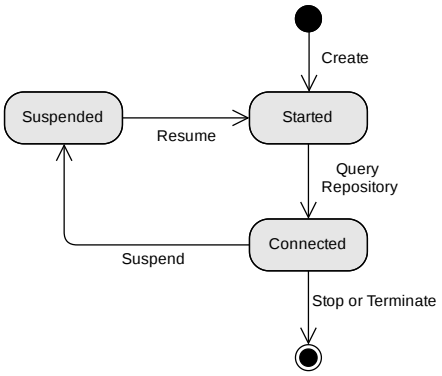


Figure 2: MirageManager — Unikernel lifecycle.

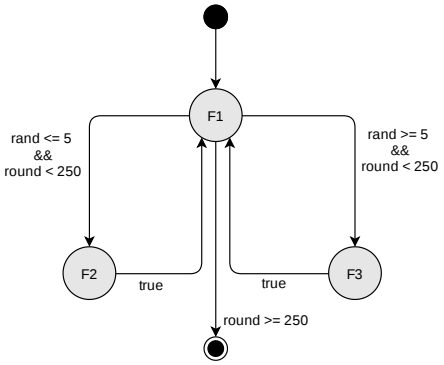


Figure 3: Checkpointing execution graph.

To enable the application to write its execution state to Unimem, we translate the unikernel into a series of atomic procedures, each constituting a step. Hence, we define the application workflow as a directed graph with labeled edges where each node is a computational step. Every step is identified by a unique string identifier (ID) so that the currently active step can be dumped into the store just by using its ID. Edges are guarded by expressions using the variables present in the store that determine how the control flow is directed from one step to the next.

In the case of multiple edges originating from the same computational step (e.g., execution logic branching), the program decides which one to follow by evaluating what we call *transition guards* which are conditional equations evaluated on variables stored in Unimem. Therefore, the control flow can be expressed as an adjacency matrix where each entry a_{ij} describes the transition from

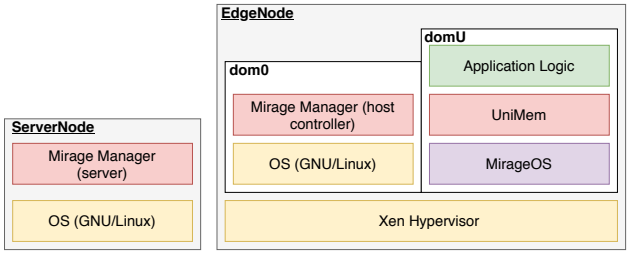


Figure 4: MirageManager components.

step *i-th* to *j-th* and its value acts as a guard for the transition. The truth condition of the transition guard is obtained from evaluating specific functions on a set of variables in the store. The first condition evaluating to *true* in a row of the adjacency matrix determines the next transition in the execution flow. One limitation is that guard functions should be mutually exclusive to avoid ambiguities in the process of selecting which transition to take at any given moment. If no guard condition evaluates to *true*, the application logic is considered to be completed and the unikernel terminates.

When the unikernel is requested to suspend or migrate, the identifier of the currently executed function is written to Unimem. As every variable used to evaluate the transition guards is stored as well, Unimem's content fully describes the application state. In fact, the current position in the graph as well as the next transition to be traversed can be inferred from the stores content only. To protect against state corruption and potential information loss, all variables belonging to an execution scope spanning multiple computational steps must be stored. This is facilitated by not using return values or parameters for the steps, but rather writing from and reading to the store. As computational steps are atomic, the information contained in Unimem is sufficient to recreate the application state after a migration.

Finally, there is no specific structure imposed on the content of Unimem by MirageManager as long as the state is serializable. Therefore, also other state information, such as the state of an object-oriented application, could theoretically be stored.

5 IMPLEMENTATION

MirageManager is implemented as a distributed system consisting of an application server developed with Express [35] and written in JavaScript. It exposes a REST API for the admin user to issue migration commands, and for the unikernel to transfer its state to the central repository. Additionally, each host wishing to use MirageManager needs to run a controller so that the central application server can communicate with the hypervisor on that machine. In Figure 4, the *ServerNode* hosts the application server while the *EdgeNode* is as device using the migration functionalities. In our implementation, we used Xen as hypervisor [36] but other options are possible, as discussed in §7.

The communication between application server and host controller is performed by remote procedure calls using gRPC [37]. The controller programmatically issues commands to Xen via the *xl* tool in order to create and destroy domains. Additionally, it uses *xenstore-write* to communicate with the unikernel guest domains. The application server API is invoked using HTTP requests for issuing commands and transferring states which are encoded using the JSON format.

Inside the unikernel, the state is stored in a Unimem object which is a singleton instantiated from an object class in our library's store module. At the core of Unimem there is a key-value store implemented using the OCaml Map module. We expressed the adjacency matrix so that the keys are the origin compute step IDs and the values are a list of records containing the tuple destination node ID and transition guard. The OCaml code snippet in Listing 1 shows the implementation of the execution flow in Figure 3.

Unimem also embeds the communication protocol functions to communicate with MirageManager and serialize the unikernel state.

```
let get_adjacency store =
  let g12 = (((Store.to_int
    (store#get "rand" (Store.VInt 0)))>= 5)
   && ((Store.to_int
    (store#get "round" (Store.VInt 0)))<=250)) in
  let g13 = (((Store.to_int
    (store#get "rand" (Store.VInt 0))) < 5)
   && ((Store.to_int
    (store#get "round" (Store.VInt 0))) <=250)) in
  let gt = ((Store.to_int
   (store#get "round" (Store.VInt 0))) > 250) in
  let assoc_adj_list = [
    ("f1", [
      {step = "f2"; condition = f12};
      {step = "f3"; condition = f13};
      {step = "terminate"; condition = gt};
    ]);
    ("f2", [{step = "f1"; condition = true}]);
    ("f3", [{step = "f1"; condition = true}]);
  ] in
  let amap = StringMap.of_seq (List.to_seq
   assoc_adj_list) in
  amap
```

Listing 1: Unimem code snippet.

In addition to the Unimem class, there are utility functions for converting the store value types to normal OCaml types in the store module. Finally, the library offers a control module providing a set of functionalities allowing the unikernel to communicate with the controller through Xenstore [38].

6 EVALUATION

For our evaluation, we selected Podman as candidate baseline to compare against MirageManager. Podman is an engine for running Open Container Initiative (OCI) containers with support for CRIU-based migration for Docker. We embedded an application with the same functionalities as our MirageOS unikernel inside an OCI container and then performed the migration tests. The application consisted of a simple numeric counter printing the current number of iterations, at every second. As we focus on the IoT domain, we decide to keep the application logic simple. At this stage, we did not consider the possibility to run and embed complex applications in an unikernel running on a contained device. We did not compare MirageManager against Docker's native container migration tool because it is still in an experimental stage and required excessive modifications and workarounds to use it in our tests. Therefore, we excluded it from our evaluation.

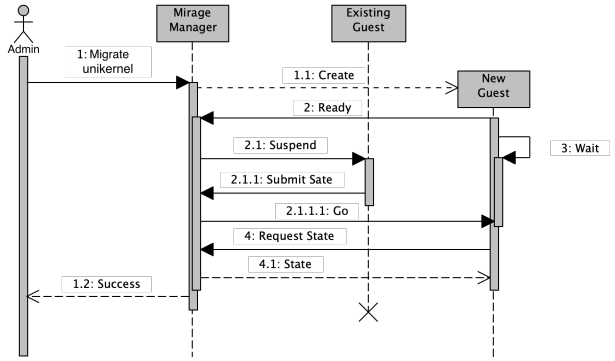


Figure 5: MirageManager migration workflow.

The migrations operations were performed between two Intel NUCs connected to the same subnet with a 100Mbps connection and running Ubuntu 18.04 with a downgraded kernel version (due to incompatibilities with CRIU). The hosts using MirageManager do not necessarily need to be on the same subnet to use the migration functionality. In this specific case, the choice was out of simplicity. To provide a quantitative analysis of our solutions against Podman, we selected four metrics:

Downtime. The time elapsed between service suspension and restart. It is the most critical metric for evaluating the performance of a live migration, as it shows for how long the service is not able to perform its task. From a user perspective, only the downtime is noticeable. As timestamps are logged for every iteration of the unikernel application logic, we can precisely measure the downtime by subtracting the two timestamps between suspension and restart.

Migration Time. It is the time required by MirageManager to perform the state migration of a unikernel between source and target machine, including the resume operation. Compared to downtime, it is a compound metric covering different steps of our migration workflow and it is calculated differently for Podman and MirageManager. For the former, a timestamp is logged both when the suspend command is issued and when Podman completes the resume command. The difference between the two values amounts to the migration time. For the latter, the first timestamp is generated at suspension time, but the second is created by the unikernel after successfully retrieving its state from the repository. We can breakdown this time interval even further. The *init* time tantamount to the kernel boot plus the initialization of a TCP/IP network connection. During the *wait* time, the new unikernel awaits for the old one to suspend and save its state. Finally, the *retrieval* time represents the time interval to query the state from the repository. These three phases show the proceeding in time of the migration workflow shown in Figure 5.

State Size. Amounts to the data needed to be transferred between hosts in order to perform the migration. This metric is less crucial for migrations happening in datacenters where bandwidth is not an issue. However, as we focus on migration in edge networks, it assumes much more relevance as the available bandwidth is limited. The state size is calculated in Bytes and for Podman is the size of the state tarball while, for MirageManager, of the JSON message embedding the state of the unikernel.

Image Size. The size of the image. Both, for Podman and MirageManager it is defined by the size of the image stored in the filesystem.

Results

Table 1 shows the migration results as the average of 200 migrations mapped to the respective steps shown in Figure 5. MirageManager provides slightly worse performance, as it takes longer than a cold migration with Podman. However, this is due to MirageManager already starting the target unikernel before transferring its state over the network. This is a time consuming operation due to the time required for MirageOS to successfully setup the network channel, especially with DHCP enabled.

Table 1: MirageManager vs. Podman

		MirageManager	Podman
Migration Time	Init [s] (Steps 1 and 2)	0.91 +/- 0.20	-
	Wait [s] (Steps 2 and 2.1.1.1)	2.61 +/- 0.45	-
	Retrieve [s] (Steps 4 and 4.1)	0.02 +/- 0.006	-
	Total [s]	3.54 +/- 0.48	1.96 +/- 0.06
	Downtime [s] (Steps 2.1 to 4.1)	0.33 +/- 0.02	1.80 +/- 0.05
	State Size [B]	79.00	195175.28 +/- 113.39
	Image Size [B]	27780862	70730752

The benefit of our approach can clearly be seen in terms of downtime. In fact, MirageManager downtime is ~**80%** shorter than Podman's. Regarding data usage, MirageManager is clearly in advantage. The container migration requires a full memory dump transfer, while MirageManager only transfers the necessary variables to preserve the application logic state. As a consequence, the amount of data needed to transfer the state between machines is more than ~**2000x** smaller with MirageManager, when compared to Podman. This is also a function of the application logic which can affect the state transfer *cost*.

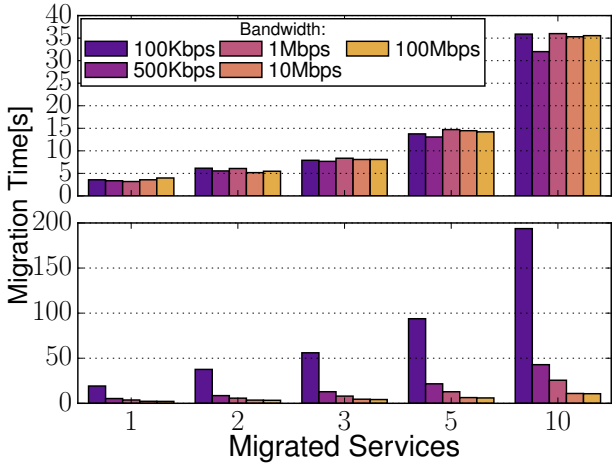


Figure 6: Migration scaling performance. MirageManager (top) vs. Podman (bottom).

An important factor when evaluating service migration in distributed systems is scalability. Therefore, we evaluated how migration with MirageManager fares in comparison with Podman when both tools perform multiple migrations simultaneously and measured the overall migration time. Figure 6 shows the overall time required to migrate multiple services in parallel. The top part of the plot shows the results for MirageManager while the bottom part shows those for Podman. In both cases, we measured the overall migration time with four different bandwidth settings. As we discussed previously, edge networks suffer from bandwidth constraints which severely impact migration operations when the transferred state is not small. This stresses the need not only to follow best-practices of service decomposition but also to reduce the state size as much as possible. For both, unikernels can be the answer.

We can gain multiple insights from Figure 6. First, MirageManager's migration time is seemingly unaffected by the available network bandwidth and it grows quasi-linearly with the amount of services migrated in parallel. The transferred state is extremely small, as we do not include the domains full memory. The same cannot be said for Podman, which is definitely suffering in low bandwidth conditions because it needs to transfer the complete memory dump as part of its migration technique. This tendency is exacerbated with the network bandwidth capped at 100 and 500 Kbps. In this case, MirageManager is up to **~6x** times faster than Podman. On the other hand, Podman outshines MirageManager as the available bandwidth increases. In fact, the latter it heavily penalized by the long *wait* time (as shown in Table 1) which is the major culprit of the long migration time. However, this is a limitation of the specific unikernel rather than our system which can be addressed in the future to drastically improve MirageManager performance.

Finally, while migration with Podman is transparent to the migrated application, MirageManager requires changes to the application logic in order to work correctly. Based on this, we state that MirageManager generally outperforms in downtime and data transfer volume cold migration with containers while offering competitive performance in terms of overall migration time.

7 DISCUSSION

In this section we discuss the limitation of our approach in relation to the our implementation and design choices.

MirageOS & OCaml. Currently, MirageManager only supports MirageOS unikernels. While MirageOS is a promising project, this results in MirageManagers biggest limitation as it forces the developer to write all code in a specific programming language (OCaml). Additionally, MirageOS unikernels compiled against Xen do not support the full set of libraries available to POSIX processes. This is due to the restricted set of libraries that have been ported to be compatible with MiniOS. However, our system could be extended and ported to work with other unikernels [39–42], which would bring more freedom in terms of available programming languages.

Virtualization. MirageManager uses Xen as hypervisor. However, in recent years we noticed how more flexible and user-friendly solutions, such as KVM [43], have received increasing attention. MirageOS is compatible with KVM and especially Solo5 [44]: a sandboxed execution environment for unikernels based on KVM. Our system could be adapted to run on top of this hypervisor, too, which would also drop some stringent requirements inherited from Xen in terms of, for example, hardware prerequisites.

Application Design. MirageManager imposes further design and implementation restrictions on a newly developed unikernel. The developer must build the application logic so that it can be serialized for a migration. This adds complexity to the development phase and requires specific knowledge of the underlying migration system. On the other hand, MirageOS unikernels benefit from a compile-time defined behavior which opens to the possibility of programmatically generating the adjacency matrix representing the execution flow. Formal proof management system like Coq [45] are natively compatible with OCaml and can help in this regard. Alternatively, we contemplate the possibility of using tools such as pre-processors in order to cope with the code modifications and language implications (e.g. return values) discussed earlier.

While these restrictions can rule out using MirageManager in some cases — what we presented is an initial prototype. Still, it is the first system enabling the migration of unikernels while managing multiple Xen hosts and their guest domains. Our design allows to easily extend the implementation to accommodate diverse hypervisors and library operating systems and, yet at an early stage, it performs competitively when compared to more mature solutions.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented MirageManager: a checkpoint-based, live migration solution for unikernels. We discussed the motivation and reasoning behind our design which stems by a surging interest for service migration at the edge. In order for unikernels to keep growing as a virtualization technology, functionalities like migration must be made available in order to extend their applicability also to stateful services. MirageManager was developed on top of MirageOS and Xen. Our evaluation showed the potential of our solution in comparison to a well established service migration approach. Nevertheless, there are limitations which open to manifold exploration paths for our future work. Improving scalability, reducing the implementation effort by automatically extracting the execution flow, testing our solution with other unikernel technologies are the first challenges we plan on tackling.

REFERENCES

[1] Brandon Amos, Bartosz Ludwiczuk, Mahadev Satyanarayanan, et al. Openface: A general-purpose face recognition library with mobile applications. *CMU School of Computer Science*, 6:2, 2016.

[2] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010.

[3] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016.

[4] Peng Liu, Dale Willis, and Suman Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 1–13. IEEE, 2016.

[5] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, pages 1–13, 2017.

[6] Yuang Chen and Thomas Kunz. Performance evaluation of iot protocols under a constrained wireless access network. In *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*, pages 1–7. IEEE, 2016.

[7] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive VM handoff across cloudlets. *Technical Report CMU-CS-15-113*, 2015.

[8] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[9] OpenVZ. https://wiki.openvz.org/Main_Page. Accessed: 2020-06-16.

[10] LXC. https://linuxcontainers.org/. Accessed: 2020-06-16.

[11] R Boucher. Live migration using CRIU, 2017. Accessed: 2020-06-16.

[12] CRIU. https://criu.org/Main_Page. Accessed: 2020-06-16.

[13] Aws lambda. https://aws.amazon.com/de/lambda/. Accessed: 2020-09-16.

[14] Michał Król and Ioannis Psaras. Nfaas: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 134–144, 2017.

[15] Vittorio Cozzolino, Nikolai Schwellnus, Jörg Ott, and Aaron Yi Ding. UIDS: Unikernel-based Intrusion Detection System for the Internet of Things. In *DISS 2020 - Workshop on Decentralized IoT Systems and Security*, 2020.

[16] Shangguang Wang, Jinliang Xu, Ning Zhang, and Liu Yujiong. A survey on service migration in mobile edge computing. *IEEE Access*, PP:1–1, 04 2018.

[17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[18] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78. IEEE, 2015.

[19] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42, 2015.

[20] Carlo Puliafito, Carlo Vallati, Enzo Mingozzi, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Container migration in the fog: a performance evaluation. *Sensors*, 19(7):1488, 2019.

[21] Carlo Puliafito, Enzo Mingozzi, Carlo Vallati, Francesco Longo, and Giovanni Merlino. Virtualization and migration at the network edge: An overview. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 368–374. IEEE, 2018.

[22] Paolo Bellavista, Alessandro Zanni, and Michele Solimando. A migration-enhanced edge computing support for mobile devices in hostile environments. In *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 957–962. IEEE, 2017.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[24] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[25] Aaron Yi Ding, Bo Han, Yu Xiao, Pan Hui, Aravind Srinivasan, Markku Kojo, and Sasu Tarkoma. Enabling energy-aware collaborative mobile data offloading for smartphones. In *2013 IEEE International Conference on Sensing, Communications and Networking (SECON)*, pages 487–495, June 2013.

[26] Flávio Ramalho and Augusto Neto. Virtualization at the network edge: A performance comparison. In *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–6. IEEE, 2016.

[27] Motoshi Horii, Yuji Kojima, and Kenichi Fukuda. Stateful process migration for edge computing applications. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

[28] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111, 2018.

[29] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, Andy Crabtree, James Colley, Tom Lodge, et al. Personal data management with the databox: What's inside the box? In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, pages 49–54, 2016.

[30] Anil Madhavapeddy and David J Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[31] Vittorio Cozzolino, Jörg Ott, Aaron Yi Ding, and Richard Mortier. Ecco: Edge-cloud chaining and orchestration framework for road context assessment. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 223–230. IEEE, 2020.

[32] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[33] MiniOS. https://wiki.xenproject.org/wiki/Mini-OS. Accessed: 2020-06-16.

[34] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 258–269, 2016.

[35] Express framework. Accessed: 2020-06-16.

[36] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[37] GRPC framework. Accessed: 2020-06-16.

[38] Xenstore-write manual. Accessed: 2020-06-16.

[39] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[40] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 459–473, 2014.

[41] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 61–72, 2014.

[42] rumprun. https://github.com/rumpkernel/rumprun. Accessed: 2020-06-16.

[43] Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page. Accessed: 2020-06-22.

[44] Solo5. https://github.com/Solo5/solo5. Accessed: 2020-06-16.

[45] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

Publication IV

Vittorio Cozzolino, Tonetto Leonardo, Nitinder Mohan, Aaron Yi Ding, and Jörg Ott. 2022. Nimbus: Towards Latency-Energy Efficient Task Algorithm for AR Services. IEEE Transaction on Cloud Computing, 2022.

Publication Summary

Widespread adoption of mobile augmented reality (AR) and virtual reality (VR) applications depends on their smoothness and immersiveness. Modern AR applications applying computationally intensive computer vision algorithms can burden today's mobile devices, and cause high energy consumption and/or poor performance. To tackle this challenge, it is possible to offload part of the computation to nearby devices at the edge. We found that this calls for smart task placement strategies in order to efficiently use the resources of the edge infrastructure. Therefore, the core of our work in this article is Nimbus: task placement and offloading solution for a multi-tier, edge-cloud infrastructure where deep learning tasks are extracted from the AR application pipeline and offloaded to nearby GPU-powered edge devices.

We focused on minimizing the latency experienced by end-users and the energy costs on mobile devices. We provide a multifaceted evaluation, based on benchmarked performance of AR tasks, shows the efficacy of our solution. Overall, Nimbus reduces the task latency by $\tilde{4}$x and the energy consumption by 77% for real-time object detection in AR applications. We also benchmark three variants of our offloading algorithm, disclosing the trade-off of centralized versus distributed execution.

Author's Contribution

I came up with the idea for the paper as a foundation for task provisioning in edge-cloud infrastructures. I have designed, implemented, and evaluated the entire system. Leonardo Tonetto supported the design phase and offered valuable insights for the data interpretation part. Nitinder Mohan helped by providing a dataset that was crucial for the network benchmarks. Leonardo Tonetto and Nitinder Mohan also supported the writing of the final publication.

Nimbus: Towards Latency-Energy Efficient Task Offloading for AR Services

Vittorio Cozzolino, Leonardo Tonetto, Nitinder Mohan, Aaron Yi Ding, Jörg Ott

Abstract—Widespread adoption of mobile augmented reality (AR) and virtual reality (VR) applications depends on their smoothness and immersiveness. Modern AR applications applying computationally intensive computer vision algorithms can burden today's mobile devices, and cause high energy consumption and/or poor performance. To tackle this challenge, it is possible to offload part of the computation to nearby devices at the edge. However, this calls for smart task placement strategies in order to efficiently use the resources of the edge infrastructure. In this paper, we introduce Nimbus — a task placement and offloading solution for a multi-tier, edge-cloud infrastructure where deep learning tasks are extracted from the AR application pipeline and offloaded to nearby GPU-powered edge devices. Our aim is to minimize the latency experienced by end-users and the energy costs on mobile devices. Our multifaceted evaluation, based on benchmarked performance of AR tasks, shows the efficacy of our solution. Overall, Nimbus reduces the task latency by $\sim 4\times$ and the energy consumption by $\sim$77% for real-time object detection in AR applications. We also benchmark three variants of our offloading algorithm, disclosing the trade-off of centralized versus distributed execution.

Index Terms—Edge Computing, Augmented Reality, Optimization, Resource Management, Cloud Computing.

1 Introduction

Since the advent of consumer mobile devices equipped with multiple sensors and powerful chipsets, multimedia applications have garnered increasing interest amongst smartphone users. A recent study reports that the mobile AR adoption currently stands at 32%, where 54% of the respondents use mobile AR at least once per week and 36% percent several times per week [7]. Despite the increasing popularity of the technology, most current mobile AR applications often offer poor user perceived performance. The reason for this is two-fold. Firstly, object recognition and detection algorithms are a bottleneck for AR [97] as the front-end devices are often insufficiently equipped to execute them with acceptable latencies for the end user [1, 23]. Secondly, extended usage of such applications results in high power consumption, which leads to significant battery drain and overheating [35, 77].

Edge computing allows applications developers to accelerate their services' performance by offloading computationally intensive tasks to nearby powerful machines instead of the distant cloud datacenter. Latency critical applications operating on mobile devices, such as AR/VR, benefit most from the availability of the edge as they can utilize more powerful hardware, in addition to on-board processors, without traversing long paths to the cloud [88, 91]. As shown in previous research, such approaches not only allow smartphones to run multimedia applications and games with better visual quality [17, 21, 30, 34, 41, 80, 81, 93], but also enable older mobile devices (provided they are equipped with the required spatial sensors) to support such applications in the first place.

Unlike other driving applications for edge computing (e.g. smart homes), real-time multimedia applications impose much stricter constraints on offloading computations at edge devices. Since such applications need to incorporate tightly-coupled user interactions, they operate under strict delay thresholds imposed by the human vestibular system – bordering between 75ms for online gaming and 250ms for telemetry [66]. In practice, requirements for seamless interaction between the physical world and the virtual overlay are estimated to be much lower, $\sim$7ms [10, 25]. Currently, a modern smartphone can run object detection in $\sim$200ms per frame using an optimized model [78], which is some orders of magnitude off from the strict requirements of AR applications. Preserving loss of smoothness and excessive delays in applications relying on virtual environment is paramount to prevent phenomena such as motion sickness [66].

Additionally, AR/VR applications are power-hungry and can quickly drain the phone's battery [9]. The growing demand for higher precision deep learning models and increased immersiveness of the augmented experience can cost even more battery power. Chen et al. [28] show that a smartphone can spend significant portion of its battery capacity while running a mobile-optimized object recognition service. Pairing this workload with client-side rendering, network communications, and running specific AR application logic can reduce the expected battery life even further.

Considering the complexities levied by deep learning based real-time applications, it is challenging to exploit a nearby edge infrastructure in a scalable manner. Moreover, while the *cloud* has potentially unlimited resources, the same cannot be assumed for the *edge* computing paradigm. In fact, the latter is by definition distributed across multiple edge networks and hence associated with considerable heterogeneity in bandwidth and compute resources [61]. On the other hand, recent large-scale measurement studies have shown

- *V. Cozzolino, L. Tonetto. N. Mohan and J. Ott are with the Technical University of Munich, Germany.*
- *A.Y. Ding is with the Delft University of Technology, Netherlands*

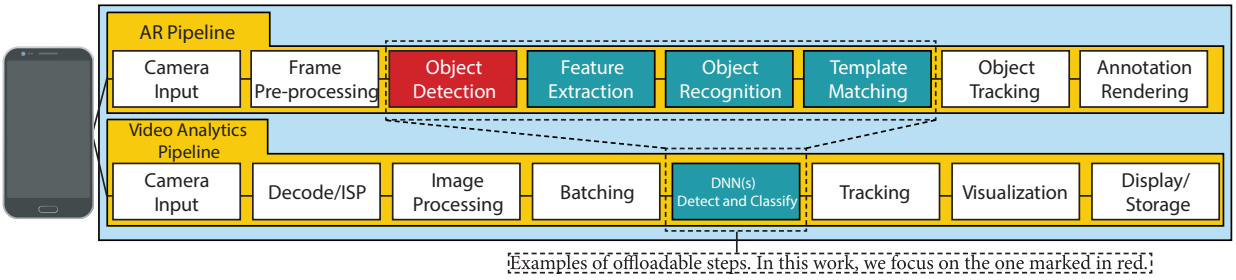


Fig. 1: Mobile applications requiring deep learning steps.

that despite the significant growth in cloud infrastructure, the network latencies from users to nearest cloud datacenters exceed the strict operational boundaries of AR applications almost globally [31, 32, 36]. As a result, we see *edge-cloud interplay* as key to extend cloud computing reach outside of datacenters, and enhance its services by leveraging an infrastructure closer to the end-users [66, 87]. We believe that effective application offloading is a crucial problem for edge-cloud computing that must be addressed when thinking at scale. For that, selecting an appropriate offloading candidate must be at the core of maximizing user satisfaction, as allocating multiple users to an already overloaded edge node can negatively impact an AR application's performance [28].

To summarize, the **motivation** behind our work is boosting the performance of mobile applications that use DNNs (as shown in Figure 1) by offloading part of their execution pipeline to the edge-cloud infrastructure. The ultimate goal is to improve the quality of experience and enable potentially new classes of applications which have strict latency constraints (such as real-time mobile VR). We approach the problem from a system design perspective and proceed by using an algorithm for resource provisioning to measure the effectiveness of our architecture.

Contributions. In this paper, we present *Nimbus*, a real-time task offloading system designed to determine an optimal task placement strategy. We aim at reducing the *latency gap* afflicting the execution of real-time deep learning models required by AR and similar applications by making use of resources offered at the network edge, at scale. We select and support the execution of mobile-optimized, object detection convolutional neural network (CNN) for AR applications, as shown in Figure 1. This shows also the pipeline for live video analytic applications which programmatically share core components of AR/VR applications and are becoming the solution to many safety and management tasks [95]. The design principles of Nimbus are devised to address three crucial constraints of target applications: (1) *latency* as a primary measure of the application QoS, (2) *battery consumption* which defines the extent of the user's QoE, and (3) *task coordination* as the role of the infrastructure in orchestrating, load balancing and distributing computation based on the users' demands. Nimbus aims at minimizing the overall mobile-to-edge latency while avoiding increasing battery consumption. Additionally, Nimbus's offloading policy ensures a balanced load distribution across the edge nodes participating in the infrastructure. Our contributions in this paper are as follows:

- We benchmark the performance of different classes of edge devices to understand their support towards real-time object detection for mobile AR.
- We devise a multi-tier edge-cloud infrastructure and propose a best-effort resource provisioning algorithm addressing the problem of serving multiple users competing for heterogeneous resources. Overall, our approach reduces task latency by $\sim$4$\times$ and the energy consumption by $\sim$77% for real-time object detection.
- We develop an edge infrastructure simulator[1] to evaluate the performance of Nimbus against other related solutions. From an empirical analysis based on extensive measurements in real testbeds, we extract the parameters of the simulator to closely mimic the realistic operations of edge devices and core network latencies.
- We develop and evaluate several variants of Nimbus reflecting both centralized and distributed execution of the task placement algorithm.

2 Related Work

The intuition of offloading computationally intensive tasks from mobile devices towards powerful servers has been explored vastly in the past decade. Originally, the offloading procedure targeted powerful datacenters in the cloud [39, 42].

With the rise of edge computing, the status quo changed drastically with new possibilities to mitigate the most prominent drawback of cloud offloading: *latency*. In fact, the introduction of cloudlets and edge envisioned a collaborative computational infrastructure where intensive tasks could be offloaded to nearby edge microservers, thus saving on access latency [20, 79]. Moreover, edge computing can also help in reducing energy consumption of mobile devices. For example, with Voltaire [27] it is proposed to perform code offloading to enables resource-constrained devices to leverage idle computing power of remote resources.

Nevertheless, edge nodes have limited computational resources, limiting the number of clients that can be served at the same time. Approaches based on offloading to the nearest edge-cloud can lead to situations where too many clients are allocated to the same node, competing for limited resources. Multiple works have focused on solving a similar problem by using either hierarchical edge-cloud architectures or load balancing among edge-cloud [24, 29, 47, 58, 62, 63, 90, 92]. In particular, MCDNN [45] developed a compiler together

1. Code and dataset are available here https://github.com/vitcozzolino/nimbus.

with a runtime scheduler to balance between accuracy and resource consumption by reasoning about on-device/cloud execution tradeoffs, while Markov decision processes [46] were used for VMs load management to reduce energy consumption in datacenters. A similar approach was proposed by Tan et al. [84] to minimize the expected response time, where tasks uploaded from mobile device are sent to an edge-cloud infrastructure and scheduled by an online job dispatching algorithm. While their method is limited – assuming a server can only process one job at a time – we instead consider parallel execution of multiple jobs. Other approaches have focused on reconfiguration of edge-clouds [51], specifically on how to optimize the placement of cloudlets in a given network. The approach of using a hierarchical edge-cloud infrastructure has been proposed already by Tong et al. [85] to efficiently handle the peak load and satisfy the requirements of remote program execution. Recent work from Braud et al. [26] introduces a task allocation algorithm based on a latency model leveraging multipath computation to offer multiple resources in parallel. The key difference from the these approaches is that our system tackles the problem of *parallel* tasks execution offloaded to the same edge device while they focus on sequentially placed workloads. Additionally, previous solutions focused only on latency (computational and/or communication) without factoring in mobile energy consumption in the offloading decision. Finally, many scheduling algorithms translate task complexity in the number of CPU cycles required for its execution eventually combined with other parameters such RAM, disk, and bandwidth [38, 53, 98]. We instead focus on GPU workloads and their performance variance with overlapping tasks — a parameter which is seldomly explored.

While most of the previous work aimed to minimize mobile task execution time, we focus specifically on AR application offloading [57, 89]. By doing so, we gain a clear understanding of how and where a task should be offloaded since we are aware of the inherent requirements of such applications. Our scheduling algorithm focuses primarily on improving the perceived performance for the mobile user. Similar work has been done for visual applications offloading in the past. LAVEA [96] is a system built on top of an edge computing platform, which offloads computation between clients and edge nodes, to provide low-latency video analytics at places closer to the users. The work closest to ours is [72] – a framework that ties together front-end devices with more powerful backend servers to support complex deep learning tasks. However, unlike our work, the authors do not consider a multi-tier edge infrastructure and scenario where multiple users are competing for the resources offered by the infrastructure.

For mobile-cloud offloading, some work has been conducted in the past for optimizing DNNs. Kang et al. [52] and Xia et al. [94] identified how to optimally slice a model to offload only a part of it to the cloud in order to minimize either latency or energy consumption. DynO [16] is a distributed inference framework addressing several challenges, such as device heterogeneity, varying bandwidth and multi-objective requirements. Key components that enable this are its novel CNN-specific data packing method, which exploits the variability of precision needs in different parts of the CNN when onloading computation, and its novel scheduler that jointly tunes the partition point and transferred data precision at run time to adapt inference to its execution environment. Our work is inspired by those studies and strives to characterize the problem in a multi-tenant environment where resource contention is the primary issue.

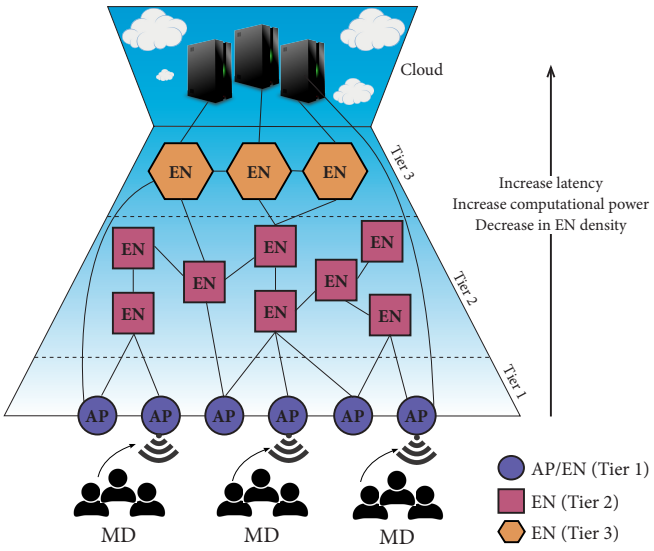


Fig. 2: Multi-tier edge-cloud infrastructure.

Recapping, Nimbus differentiates from the aforementioned research works in many ways. It tackles the problem of parallel task execution instead of makespan optimization (sequential). Our offloading solution revolves around a joint latency and mobile battery optimization procedure with a focus on GPU workloads and their scaling properties. Finally, Nimbus performance is rooted in a set of real measurements gathered from devices which are part of our envisioned edge-cloud infrastructure.

3 System Overview

Figure 2 shows the *entities* in our system – mobile devices (MD) and edge nodes (EN) interacting over the network. While the former interact with the infrastructure as users of AR applications, the latter are responsible for handling tasks offloaded by the MD. In our case, an MD is a battery-powered mobile device that can offload part of its computation to the edge-cloud infrastructure. We assume a hierarchical edge architecture where compute and caching capabilities of EN increase with increasing distance from the MD. Nodes in different (logical) layers of the edge network can be accessed via ad-hoc connections or gateways [33, 56, 60, 65, 67, 83].

The design of our edge computing infrastructure is inspired by networks like Eduroam[2]. The deployment of Eduroam is widespread as it can be found outside academic facilities, e.g., libraries and study centers. While such a network (currently) only offers Internet access to clients, we acknowledge its capabilities to support an edge computing infrastructure due to the presence of multiple connected

2. https://www.eduroam.org

networked resources capable of running computations on behalf of the connected users. We logically divide the network into three layers – each one offering different capabilities and, as we approach the core of the infrastructure, latency and computational capacity of the resources increases. Conceptually, the architecture proposed by Tong et al. [85] and Mohan and Kangasharju [64] come closest to ours and we use them as point of reference in our system design.

Tier One Edge Nodes (T1-EN). The outer-most layer (denoted by blue circles in Figure 2) is a set of augmented access points (AP) or base stations with minimal compute capabilities. We assume these APs to be either equipped with (or directly connected to) an embedded device with low-end GPUs, e.g. NVIDIA Jetson Nano or Intel NCS2. Resources in this layer act as entry points to the network, offering limited computation in addition to standard routing and connectivity functionalities.

Tier Two Edge Nodes (T2-EN). T2-EN (denoted with squares Figure 2) form the second layer of our multi-tier edge cloud infrastructure. Logically these devices can be viewed as backbone routers co-located close to T1-EN. However, unlike T1-ENs, T2-ENs posses more computational power and network bandwidth that allows them to serve multiple users in parallel. An example of T2-EN resources in the real world is a mid-range micro-server equipped with a discrete GPU.

Tier Three Edge Nodes (T3-EN). The core of our architecture comprises of T3-EN (shown as orange hexagons) that are powerful servers equipped with multiple GPUs, offering the most significant computational power of all layers. The capabilities of T3-EN are analogous to traditional cloud datacenters, both in terms of the number of users that can be served in parallel and network bandwidth connecting servers within the layer. However, due to their proximity to the network core, the network latency incurred to access the resources in this layer is the highest amongst edge infrastructure.

We consider a system where a mobile device hosting an AR application can offload component tasks in the pipeline (e.g. those requiring deep learning) to the edge infrastructure. Considering the inherent heterogeneity that exists in the infrastructure — different hardware capabilities, network latency to server, task requirements etc. — an effective task offloading strategy is required ensuring that the application performance meets the required expectations. Additionally, we assume that ENs in our system are managed resources and can communicate/exchange details regarding their current processing load with other ENs. This assumption roughly resembles the current state of resource management in cloud datacenters and it allows our task offloading algorithm (presented in the following section) to have fresh information regarding the edge network state.

4 Task Offloading at the Edge

We consider a system where a controller estimates the feasibility of offloading a task proposed by a mobile device to the edge infrastructure. Since our objective is to showcase the effectiveness of our offloading solution, we start by considering a centralized controller located in the cloud. Later in the paper (§6), we design a distributed and hybrid variant of our offloading mechanism and compare the operational differences of all approaches. Figure 3 shows a high-level concise workflow representing the MD-Controller interaction. The infrastructure is composed of **N** interconnected and heterogeneous ENs, which, based on their computing capacity, can serve several concurrent tasks. An MD can offload its task via T1-EN, which act as gateways to the infrastructure.

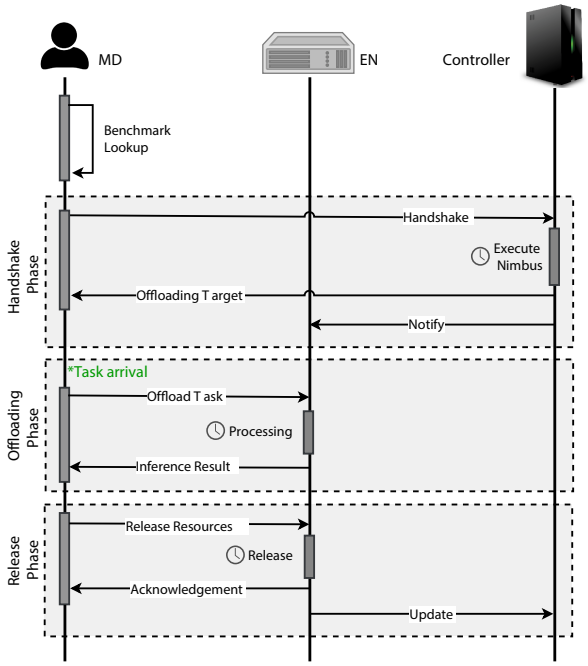


Fig. 3: MD-Controller Workflow.

Before entering the *handshake* phase, the MD performs a one-time procedure called benchmark lookup. Normally, games and other multimedia applications run benchmarks to estimate their runtime performance in order to tune and set configuration parameters. Similarly, there are tools to profile deep learning models on mobile devices [50]. In our model, we assume that benchmarked results for each MD are uploaded to a repository that is looked-up by the system to identify MD's capabilities. Afterwards, the *handshake* procedure begins and the MD exchanges with the infrastructure controller its requirements in terms of deliverable performance (in FPS and battery consumption). In the *offloading* phase, the MD connects to the network to offload and it receives a list of offloading candidates from the controller obtained by running Nimbus (details about the algorithm logic will be provided in §5). Then, the MD will interact with the selected EN until required by the underlying application. Finally, in the *release* phase, the resources booked for the MD on the EN are released, and the controller is notified. The Nimbus offloading decision is based on minimizing deep learning based task latency (*i.e.*, maximizing FPS) as it directly affects the QoE for mobile AR applications.

Offloaded Tasks. As shown previously in Figure 1, AR/VR applications (especially games) can be decomposed into subroutines executed at each rendering step [97]. Some of these steps are not tied to the application logic and are

TABLE 1: List of parameters used by the algorithm.

Term	Description	Unit
d_i	Amount of data transferred by the *i-th* MD	KB
BW_{ij}	Bandwidth between *i-th* MD i and *j-th* EN	Mbps
TET_j	Inference time on the *j-th* EN	ms
TEC_i	Local energy execution cost for *i-th* MD	mJ
q_j	Queuing time at *j-th* EN	ms
w	Transmission module power	mJ/ms
ϵ_t	Latency threshold	ms
ϵ_b	Energy budget	J/s
RTT_m	RTT matrix	ms
κ, α, β	Additional coefficients (described in § 5)	—

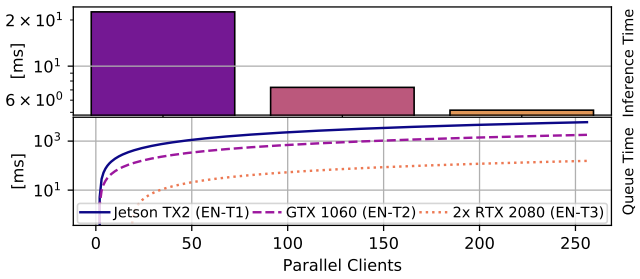


Fig. 4: Inference and queue time for the three EN tiers.

perfect offloading candidates. Let us take the example of tracking-by-detection principle [19, 55, 71] for object tracking. The principle requires that the object is detected in the first and all subsequent frames. The object is tracked simply by associating detection results to form target trajectories. This is a necessary component in all AR applications where smooth integration with real world is paramount. While tracking requires sophisticated application logic to interpolate objects positions across frames, detection is oblivious to past executions and depends only on the latest frame. Therefore, object detection is a prime candidate for offloading to the edge. In practice, a stream of pictures can be sent by the MD towards the target EN for processing. Even if the EN becomes unresponsive, the MD can switch to executing the task locally so that the underlying offloading process is transparent to the end-user who would experience no interruptions in the service. We consider each task submittable to the infrastructure as *atomic* (i.e., indivisible and uninterruptible). In this work, we focus on stateless tasks that are resilient to the loss of connectivity due to their independence from past transactions. However, our solutions proposed in this paper can be extended to stateful tasks as well with proper synchronization mechanisms. That, we leave them for future work as hereby we concentrate our efforts on the offloading strategy and algorithm formulation.

Our problem formulation assumes that MDs offload tasks to the system in bulks, which translates into a constant, worst-case arrival rate. This allows us to devise a solution that does not assume any prior knowledge about the offloaded task makespan nor use it to optimize the decision making process. It is not realistic for an MD to know in advance for how long the user will run the application (*e.g.*, minutes to hours). The only information available are inference time of the DNN task and its energy cost (estimated during the benchmarking phase shown in Figure 3). Therefore, we optimize the resources allocation in a *maximum concurrency scenario* – where all the MD are concurrently using the infrastructure and all resources, from network bandwidth to compute, must be shared.

In a practical scenario, tasks can have different complexity and requirements. For simplicity, we select a class of tasks for which we provide execution time distributions for the device executing them. We used the NVIDIA Triton [14] suite to benchmark MobileNetv2, a common CNN-model central to image classification tasks, with an increasing number of clients. We run benchmarks on three device types, each representative of the different tiers of our multi-tier edge infrastructure. Figure 4 shows the results we gathered in our experiments and specifically the inference and queue time for different EN tiers. More details will be discussed in §6.

Objective 1: Minimize Latency. The total task latency consists of transmission time Lt_{ij} between the *i*-th mobile device and the *j*-th edge node, plus the execution time Le_{ij} of the required tasks at the *j*-th node. Transmission time depends on the network bandwidth BW_{ij} and on the amount of data d_i sent by a mobile device. Furthermore, this communication delay can be negatively affected by multiple clients interacting with an EN if they share the same access medium (e.g., WiFi). Hence, a fair queuing best-effort communication model is assumed where each client connecting to an EN perceives a connection bandwidth equal to R/N, where R is the total offered data rate and N the number of active users.

Execution time represents the amount of time an MD has to wait in the processing queue before its request can be served, i.e the time to execute a task (TET_j) plus the GPU queuing time (q_j) on the *j*-th EN. Queue time can grow substantially depending on the EN's capabilities and the number of concurrently served clients. Figure 4 shows inference and queue time for each edge device tier and the number of users concurrently using the device. As expected for T1-EN, the queue time increases with the number of served clients due to the limited hardware capabilities of devices in this tier. On the other hand, powerful discrete GPUs found in expensive workstations can handle many more clients with a minimal queue time penalty. Another key insight from Figure 4 is that unlike inference time, queue time is heavily influenced by the number of parallel users and is a primary variable to model highly concurrent scenarios. Moreover, ENs equipped with powerful GPUs incur a queuing penalty only after concurrently serving many MDs, as shown for the T3-EN in Figure 4. Eventually, this leads to a point where even a powerful EN can not meet the QoE requirements of the MDs.

Objective 2: Reduce battery consumption for the MD. When mobile phones receive or transmit data, they consume energy depending on the network bandwidth and the amount of data to be transferred. Additionally, in real scenarios, wireless mobile devices often experience high variances in link quality [36, 68], directly affecting the data transfer latency and the final energy consumption. When offloading or accessing cloud resources, it is important to take into account the additional delay introduced by the network load pattern as they change throughout the day [59]. Therefore, network conditions for mobile devices experience high variance, and

Device	Name	Inference	
		Energy Cost	Time
OnePlus 5T	Mobile_A	182 mJ	154 ms
OnePlus 3	Mobile_B	318 mJ	116 ms
Redmi Note 4X	Mobile_C	268 mJ	190 ms

TABLE 2: Mobile inference time and energy cost for MobileNetv2.

narrowing down to a single energy consumption model for all kinds of mobile devices in all network conditions is very challenging. In this paper, we build on top of previous work from Xia et al. [94] and Kang et al. [52] to express the energy cost of transferring data B_t as a function of the transmission module power w and the overall transmission time L_t as shown below:

$$L_t = \frac{d}{BW} + RTT \quad (1)$$

$$B_t = L_t \times w \quad (2)$$

where *d* is the amount of transferred data, *BW* the upload bandwidth and *RTT* the network round-trip time.

We follow Xia et al. [94] and define three classes of mobile devices, each one with different hardware resources and power consumption profiles. We benchmark the performance of all three device classes for executing MobileNetv2. For Mobile_A and Mobile_B class, we use a single, CPU core while for Mobile_C we use 8 CPU cores. The energy cost and inference time achieved by all classes is shown in Table 2. In all cases, no model partitioning was applied. Also, the amount of energy spent to execute inference locally on the mobile device allows us to compare the cost of offloading the task against running it locally. While many contributions model both network transfer and mobile inference energy cost [43], we favor the approach described above due to its comprehensiveness and precise results — especially for the object detection task we focus on in our study.

Mathematical formulation. Assume that the *i*-th task is executed by *j*-th EN, the task latency and battery consumption incurred by the device can be formalized as:

$$L_{ij} = (Lt_{ij} + Le_{ij}) = \left[(\frac{d_i}{BW_{ij}}) + RTT_{ij}\right] + (TET_j + q_j) \quad (3)$$

$$B_{ij} = Bt_{ij} = Lt_{ij} \times w \quad (4)$$

We ignore the downlink cost for the energy consumption calculations as we assume it to be negligible when compared to the uplink, especially for object detection applications. While the input can be an image of arbitrary size, the output are bounding boxes of comparatively smaller in size for which the network transmission has a negligible energy cost. Therefore, when a task is offloaded, both its latency and mobile energy consumption are affected by the process of communicating with the edge infrastructure. In other cases, the task is running locally and its execution latency and energy consumption are described in Table 2.

Based on the system described above, we define the task assignment problem as *selecting an EN for assigning a task to minimize latency (L) and battery consumption (B) for the mobile device*. The problem translates into a multi-objective optimization problem with two objective functions in the form of $\min g(L(\vec{x}), B(\vec{x}))$ with $\vec{x} \in X$ and X the space of feasible decision vectors. In our case, we focus on identifying a set of Pareto optimal solutions which, by definition, cannot be improved in any of the objectives without degrading at least one of the others.

To solve for both latency and battery consumption, we make use of an approach called *scalarizing*. Scalarizing is an *a priori* method that allows us to formulate a single-objective optimization problem such that optimal solutions to it are Pareto optimal solutions to the original multi-objective optimization problem [49]. In our case, it would lead to the following reformulation of the problem: $\min g(L(x), B(x), \phi)$ with $x \in X_\phi$ and X_ϕ set depending on the vector ϕ. Of the multiple scalarization techniques, we adopt the ϵ-constraint method [37] to reformulate the multi-objective optimization problem by just keeping one of the objectives and restricting the rest within user-specified values (which fits our scenario). Based on the system described before, the offloading problem demands us to identify the best EN to run a user submitted task to minimize the experienced task latency while not violating the stated constraints. Mathematically, let $x_{ij} \in \{0, 1\}$ denote the case when the *j*-th EN serves the *i*-th device. We express the ϵ-constrained latency minimization problem as follows:

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} x_{ij} L_{ij}(p) \quad (5)$$

$$\text{subject to} \sum_{i=1}^{N} x_{ij} = 1, \forall j \in M, \quad (6)$$

$$L_{ij} \leq \epsilon_t, \forall j \in M, \quad (7)$$

$$B_{ij} \leq \epsilon_b \equiv TEC_i, \forall j \in M, \quad (8)$$

$$x_{ij} \in \{0, 1\}, \forall i \in N, \forall j \in M \quad (9)$$

where N and M are the set of mobile devices and EN, respectively, and with $\mathbf{p}$= $\langle d_i, BW_{ij}, TET_j, RTT_{ij}, q_j \rangle$ vector containing part of the parameters shown in Table 1. Equation 5 is our objective function. Equation 6 and 9 limit each MD to offload its task to as single EN, at most. Equation 7 and 8 are formalization of the latency and energy consumption constraints limiting the feasible solution space.

Constraints. The ϵ_t represents a predefined latency threshold after which offloading computation does not benefit the mobile device. The value covers both the transmission time and remote execution of the task. Depending on the mobile devices' requirements, ϵ_t can be a different value reflecting the specific user or application needs. Therefore, the threshold value depends on many factors, e.g., FPS requirements of the offloaded task. ϵ_b represents the battery consumption threshold, exceeding which offloading the task becomes too expensive in terms of energy. Fundamentally, ϵ_b depends solely on the cost of running the task locally (TEC_i) on the *i*-th device. This constraints are a function of the MD capabilities. For example, a powerful MD will have a much lower value for ϵ_t and, potentially, ϵ_b as it can complete locally its task quickly and efficiently (in terms of energy cost).

Algorithm 1: Nimbus allocation algorithm.

Input : Refer to Table 1.
Output: Best offloading target for the *i-th* MD.

```
// Warmup
E⃗N_r ← FilterAndMinimize(AP, RTT_m, ε_t)
E⃗N ← LookAheadLoad(E⃗N_r, κ)
// Core
for EN_j in E⃗N do
    L_ij = (Lt_ij + Le_ij) =
      [(d_i / BW_ij) + RTT_ij] + (TET_j + q_j)
    B_ij = Bt_ij = (Lt_ij × w)
    if L_ij ≥ ε_t or B_ij ≥ ε_b then
        Drop(EN_j, E⃗N)
    end
end
if E⃗N ≠ ∅ then
    for EN_j in E⃗N do
        return arg min[α * L_ij/ε_t + β * load_j/maxload_j]
    end
else
    // Failover
    cloud ← FindClosest(ε_t)
    if L_cloud ≤ ε_t and B_cloud ≤ ε_b then
        return cloud
    end
end
return ∅
```

Src \ Dst	T1-EN				T2-EN				T3-EN		
	EN_0	...	EN_{n-1}	EN_n	EN_0	...	EN_{n-1}	EN_n	EN_0	...	EN_n
$T1-EN_0$	c	...	...	...	...	...	...	...	...	...	...
...	...	c	...	...	...	...	...	...	...	...	...
$T1-EN_{n-1}$	...	...	c	...	...	...	...	...	...	...	...
$T1-EN_n$	...	...	...	c	...	...	...	...	...	...	...

TABLE 3: RTT matrix structure.

5 ALGORITHM

Following the ϵ-constrained approach in §4, we are able to re-construct our optimization problem in convex form that we solve using a meta-heuristic. The adopted search strategy for our meta-heuristic is inspired by the *hill climbing* algorithm [3] that is widely used due to its effectiveness and simplicity in different convex optimization problems (*e.g.*, artificial intelligence) for which it can provide the optimal solution [74]. Algorithm 1 describes Nimbus task offloading approach.

Nimbus operation is divided into three phases: *Warmup*, *Core*, and *Failover*. The *Warmup* phase identifies a list of ENs that are accessible from the AP the device is connected to and are the best candidates to offload computation. In the *Core* phase, the algorithm calculates the latency and battery cost for offloading to each EN in the list using the formulation described in §4. Afterwards, it selects the best EN based on the balance-ensuring allocator. In the *Failover* phase, if the algorithm failed to find a suitable EN for offloading the task, it looks for a cloud server that best satisfies the latency and energy consumption constraints of the task.

Algorithm 2: LookAheadLoad procedure.

Input : $\vec{EN}$, exploration coefficient κ, ϵ_t.
Output: List of compatible EN.

```
1  compatibleEN = ∅
2  for EN_j in E⃗N do
3      if devicesList_j ≠ ∅ then
4          mnl = arg max deviceNetworkLatency_j
5          if size(devicesList) ≪
             maxServableDevices_j(ε_t − mnl) then
6              compatibleEN ← EN_j
7          end
8      else
9          compatibleEN ← EN_j
10     end
11 end
12 if kappa ≡ 0 then
13     return compatibleEN
14 else
15     return randomSet(κ, compatibleEN)
16 end
```

Warmup Phase: To start off, Nimbus identifies a list of EN candidates for offloading the task. The function *FilterAndMinimize* extracts the set of ENs reachable from the AP to which the mobile device is connected. For reducing the search space, Nimbus filters out all ENs for which the network latency or the queue time is already greater (or equal to) the maximum threshold ϵ_t for the *i-th* mobile device. Subsequently, *LookAheadLoad* removes those EN candidates which are already close to their critical mass and serving another MD would violate the ϵ_t constraint. In fact, whenever we offload a task to an EN, the queue time increases for all the other tasks. As we can quantify this *domino effect* (discussed in §4), it is possible to use the MD experiencing the highest network latency as a reference point. If, for such device, we violate the task latency constraint, that EN is excluded from the list of viable offloading targets. Algorithm 2 describes the procedure in details. The parameter κ controls the search space by setting an upper bound to the number of ENs we want to consider. In our evaluation, κ will be used as a tradeoff parameter between convergence time and the solution's goodness. The output of the *Warmup* phase is a list of ENs that are passed to the next phase of the algorithm.

Core Phase: As the name suggests, this phase is the core of the algorithm as it identifies the best offloading target by solving the minimization problem defined in §4. For each of the candidate ENs collected by the *Warmup* phase, Nimbus calculates the execution latency and battery consumption for offloading the task. We then use these estimates in the optimization step to identify which ENs respect the latency and battery constraints and avoid overloading the EN. This step is necessary for an effective task offloading at the edge as any new mobile device allocated to an EN impacts the QoS of all the other device being served by that EN. We assume that MDs do not change their requirements after being offloaded. If, in case they do, the device needs to resubmit the updated requirements triggering a new schedule by

the algorithm. The α and β coefficients strike a balance between minimizing the latency for the MDs and avoiding infrastructure overload. If latency optimization is the only objective for the infrastructure's orchestrator, it can easily achieve it by setting β to zero. In our evaluation, we set α to 0.7 and β to 0.3 to strongly favor latency optimization rather than balancing the infrastructure load. Other combinations can be used depending on the specific optimization goals and on the infrastructure capacity. Additionally, $load_j$ and $maxload_j$ represent the current and maximum load in terms of devices for the *j-th* EN, respectively. We calculate the latter using an inverse formula of the queue time growth, which we omit for brevity.

Failover Phase: The final phase of Nimbus is optional as it is only reached if the algorithm is unable to find any suitable candidate in the edge infrastructure that can meet the mobile device requirements. In this phase, *FindClosest* identifies the best datacenter (in terms of network RTT) for offloading the task. We do not assume any prior knowledge of the compute and hardware capabilities of the target datacenter. Instead, we assume constant execution latency for the cloud, making network RTT the main discriminating factor.

Finally, if the *Failover* step fails, the mobile device fallbacks to local execution and exits the scheduling algorithm.

6 Measurement and Evaluation Setup

To evaluate Nimbus's performance in realistic settings, we conduct several experiments and measurements to collect data concerning multiple variables of the algorithm. In this section, we explore and analyze all facets of our algorithm, namely *network latency*, *inference and queuing time*, and *specifications of MD and T1-EN*. Note, however, that we do not simulate or model network flows. From the network perspective, we elevate our point of view so that all the consequences of routing queues, path selection, and network connection fluctuations are reflected solely by the network RTT. We delve deeper into the consequences of our choice in § 8.

Network latency. As mentioned in § 3, we target an academic network infrastructure like Eduroam. At the time of writing, no network latency datasets were available for such a network. Nevertheless, to provide a meaningful distribution of the network latency across different layers of the infrastructure, we followed two approaches. The first approach focused on measuring network RTTs targeting some of our devices connected to the Eduroam infrastructure. We performed measurements from three vantage locations: overseas (connecting USA to Europe), from a different city (∼20 miles away), and directly connected in the same subnet. By analyzing these data, we generated three probability distributions, one for each EN tier.

We utilized two publicly available RTT datasets from two p2p-based networks: Seattle [11] and PlanetLab [8]. The dataset is publicly available at [4]. In order to assign network RTT to each EN, we identified three latency classes through k-nn clustering and subsequently generated the respective distributions, shown in Figure 5. The distributions were then used to generate relative RTT matrices (Table 3) that we feed to our solver. The row and column of the matrix represent an AP and EN in the network respectively. The values of the

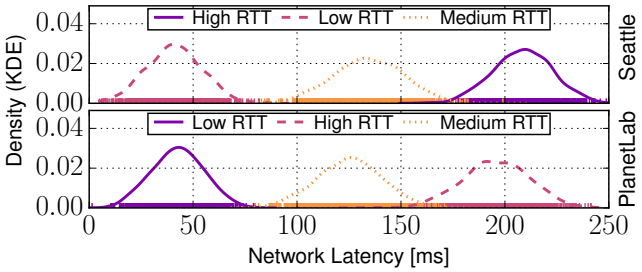


Fig. 5: Latency distributions for selected datasets.

matrix represent the network RTT to reach any of the EN from an AP. As the probe's data are anonymized, we do not have information about the relative distance of the nodes or their location.

Since our results from Seattle and PlanetLab datasets were almost similar, we only estimate latencies within our edge infrastructure using numbers from the Seattle dataset in § 7. As described in the *Failover* phase, MDs are allowed to connect to a cloud server if the performance offered by the edge network is not satisfactory. To estimate user latency to the cloud, we utilize our large-scale `ping` measurements from 3200+ RIPE Atlas probes [82] to 101 datacenters operated by seven major cloud providers globally. Our measurements over five months resulted in ≈3.2M datapoints spanning several GBs [31]. We make our dataset publicly-available at [40].

Inference and queuing time. To measure the computational cost of the task, we selected three different devices: an NVIDIA Jetson TX2, a laptop with an NVIDIA 1060 GTX, a micro-server with 2x NVIDIA 2080 RTX. We used the NVIDIA Triton [14] suite to benchmark MobileNetv2 with an increasing number of clients. Finally, as shown in Figure 4, we extracted the inference and queuing time. While the former remains constant regardless of the number of users, the latter instead, grows quasi-linearly with the number of clients. Note that this also depends on the amount of model instances loaded into the memory, as GPUs with more available VRAM can host more models in parallel, effectively boosting the overall performance by being able to concurrently serve more clients in parallel. T3-EN nodes have plenty of VRAM but this is not the case for T1-EN which might only be able to load concurrently a handful of models.

MD and T1-EN setting. The MDs are assigned hardware specs based on § 4. For simplicity, we uniformly distribute the total mobile devices across the three available hardware specs. The ratio of APs that are also T1-EN nodes is variable and depends on the experiment we run. However, for each AP, the maximum nominal Wi-Fi bandwidth is set to 300 Mbps. We assume that all devices connected to an AP experience the same connection quality apart from the effective bandwidth. Additionally, we do not account for any transmission-related issues that could negatively affect the signal.

We extrapolate data from the publicly accessible Leibniz-Rechenzentrum (LRZ) dataset [5, 6] to assign a location to each AP in the edge-cloud network plus their respective loads in terms of connected MDs. We extracted nine months'

worth of network association data of public buildings and networks from the LRZ dataset. This contains over 4500 access points scattered across ∼450 buildings. The data are aggregated in 15 minutes slices, which we use as MD-batches in our system (see § 4). We partitioned the dataset in different approaches, described further in the following section. We also compare the performance of several variants of our algorithm in the evaluation and discuss the tradeoff between convergence speed and efficiency of Nimbus.

7 Results

The results presented in this section cover two parts: (*i*) performance gain on MD and (*ii*) algorithm capability. We ran multiple experiments in different conditions (summarized in Table 4), highlighting different characteristics of our algorithm. Due to space constraints, we select a set of scenarios to showcase out system capabilities.

(A) Scalability & Performance. We first analyze the effective task latency and energy benefits[3] of Nimbus for processing tasks offloaded by MDs. We select four combinations of edge infrastructure and MDs, plus we set the required FPS threshold to 15 (frame interval ∼66.6 ms). We select four configurations where the number of connected MDs are 500, 1000, 2000, and 4000. Figure 6 depicts distributions of total task execution time and saved energy (per 1 second, or 15 frames) for 100 simulation iterations.

Even in the worst case (left panel of Figure 6), the expected task latency achieved by Nimbus is ∼**2**× lower than running it locally on the fastest MD in our dataset (see Table 2 in § 4). From a performance standpoint, this offloading strategy can boost deep learning based applications and increase the quality of experience for its end-users. As the number of MDs increases, the performance proportionally decreases. With more congestion and tasks offloaded, the delivered performance drops, as multiple MDs use the same EN and influence each other's execution time by increasing the overall queuing time. This saturation behavior is mirrored by the MDs allocation ratio. Figure 7 shows the percentage of mobile devices served by the edge infrastructure, for four different configurations of ENs shown in Table 4-(A). With an increasing number of users, the edge resources tend to saturate more quickly, forcing most of the mobile devices to run their computation locally or utilize the cloud. We find that, with the largest infrastructure used in our experiments (constituting 4000 MDs), roughly 75% can offload to the edge. Conversely, only 25% utilized the edge in our smallest infrastructure configuration.

Task offloading also allows MDs to save energy (right panel of Figure 6), reducing the power consumption in all cases. These results are significant as battery consumption is hugely relevant for high user satisfaction and retention [99]. Offloading tasks from more modern phones will lead to lesser energy savings due to their more efficient hardware components, decreasing the battery cost for running deep learning tasks. However, our results show a non-trivial margin of gain in offloading using Wi-Fi to the edge infrastructure. Even if we consider the most power-hungry smartphone in our dataset and the average energy saving in the worst-case, Nimbus still consumes ∼**77%** less battery. Note that using a mobile connection (*e.g.*, 4G) alongside task offloading leads to different results, which we discuss in the next experiment.

3. We calculate energy benefits by comparing the energy cost for offloading the task to running it locally at the MD.

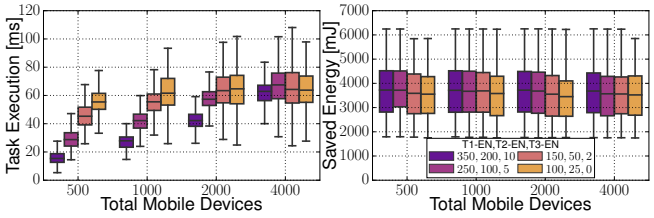


Fig. 6: Task latency and energy saving in various setups.

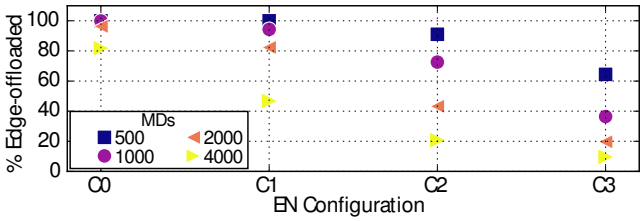


Fig. 7: Fraction of task offloaded to the edge (see Table 4).

Takeaway 1. The offloading strategy of Nimbus can boost deep learning based applications and increase the perceived performance for its end-users. The expected task latency achieved by Nimbus is ∼**2**× lower compared to the fastest MD in our dataset. Additionally, MDs consume up to ∼**77%** less battery when offloading with Nimbus.

(B) Full dataset. For this test, we run our algorithm on the entire nine-month LRZ dataset but limited to the top five most-populated buildings. Additionally, we set a minimum threshold of 30 MDs to simulate a reasonable load on the infrastructure. We fix the other parameters to values shown in Table 4. This experiment provides a broader view of the algorithm performance over an extended period with a fixed-sized edge infrastructure.

The time-series in Figure 8 shows the task execution latency (top) and MD density (bottom) for the entire nine-month period. To obtain these results, we progressively feed our algorithm with 15-minutes snapshots of MD densities from the LRZ dataset for the selected set of buildings. We further group the results by months for ease of readability. The selected buildings are part of a university campus, therefore, they exhibit a lower concentration of MDs during summer holiday period (July-September). Consequently, between October and January, the higher delivered task latency grows with the concentration of users connected to the network. However, due to the fairly low number of devices (between 30 and 500) and the generous size of the edge infrastructure, the median task latency is low. For example, the fastest MD in our dataset has a local inference time of 116 ms, which is almost **4**× higher than the average task latency that our edge infrastructure can deliver.

Figure 9 shows the relationship between the amount of saved energy for the MD and task latency, giving additional insights compared to Figure 6. For this plot, we calculated

Configuration	Edge-cloud Infrastructure					Mobile Devices				FPS	RTT Dataset
	C#	AP	T1-EN	T2-EN	T3-EN	Density	Low-End	Mid	High-End		
(A) Scalability & Performance	C0 C1 C2 C3	4371 4371 4371 4371	350 250 150 100	200 100 50 25	10 5 2 0	(500, 1000, 2000, 4000)	33%	33%	33%	15	Seattle
(B) Full Dataset	—	182	182	30	3	≥ 30	33%	33%	33%	15	Seattle
(C) Nimbus Baseline	—	4371	100	50	2	1000	33%	33%	33%	10	Seattle
(D) Nimbus Variants	—	4371	100	50	2	1000	33%	33%	33%	10	Seattle

TABLE 4: Evaluation settings.

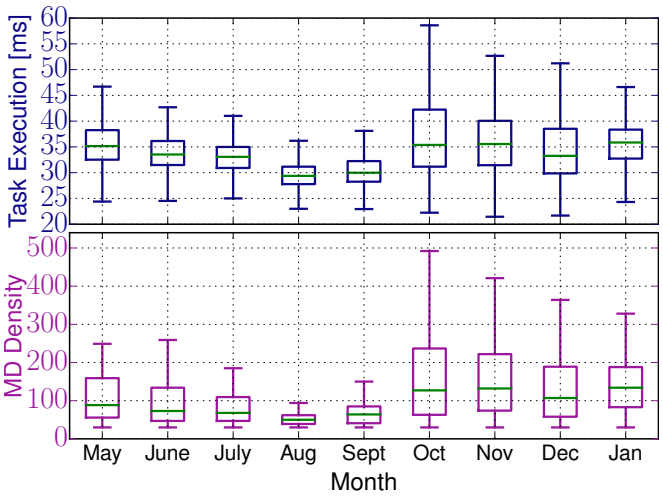


Fig. 8: Mobile devices and task latency for the LRZ dataset [6].

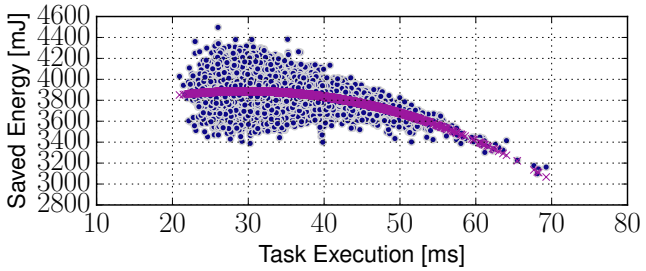


Fig. 9: Energy saving in relation to task execution time.

the energy consumption when the algorithm allocates all the MDs. The trend line demonstrates that with a higher task latency, we tend to save less energy. The leading cause can be a longer transmission time due to lower available uplink bandwidth. As an additional observation, we note having a static edge-cloud infrastructure might not always be the best option as the MD density changes at different times of the year. We hypothesize the possibility of a dynamic edge-cloud infrastructure where EN can be added dynamically in response to an increased density and demand of MDs. This would be similar to cloud computing, where resources are managed on-demand.

Finally, we analyzed the overall MD allocation ratio for the slice of data extracted from the dataset. Notice how 28.3% and 51.6% of total the MDs are allocated to T1- and T2-EN, respectively. The reasons for this can be manifold. Firstly, the number of T1-EN exceeds other tiers in our infrastructure and offers the lowest network RTT which compensates for the longer execution time. However, due to their limited resources, they can only serve a handful of MDs. T2-ENs, on the other hand, are more powerful and strike a good balance between scalability and network latency. Only 18% of the MD were offloaded to T3-EN as they offer low computation time but at the expense of *longer* network RTT. We remind that the MD population is small for this experiment. In fact, increasing the MD density pushes the algorithm to allocate more on T3-EN, as it is the only class of edge nodes capable of scaling efficiently without hindering performance. Finally, 1.1% of the MD ran the task locally, and the remaining 1.1% used the cloud. In the next section, we investigate how infrastructure size and the amount of MDs affect these ratios.

(C) Nimbus Baseline. For the baseline comparison, we evaluate our algorithm against a greedy version for 100 repetitions. Additionally, we also compare against a scenario where only cloud datacenters are available as offloading candidates, and MDs access them via either WiFi or 4G. We do not compare directly against other related algorithms (discussed in § 2) as our task allocation is fundamentally different from these approaches. Unlike related approaches, we do not rearrange and serialize the tasks to minimize the makespan but allow them to execute in parallel. For a fair comparison, we set side by side our approach with variants of Nimbus, which closely mimic the core ideology of related task offloading algorithms.

The *greedy* variant of Nimbus is fundamentally selfish: it selects the most profitable offloading candidate regardless of the possible performance degradation for the other MDs. While the standard version of the algorithm will use an unlimited search space, the greedy one will instead favor a quicker, local solution that minimizes network latency. This approach is typical of greedy algorithms that make the locally optimal choice at each stage [22]. We exploit the exploration parameter (κ) to limit greedy Nimbus's search space. The parameter also allows us to force the algorithm to produce the best offloading target from the network latency perspective and ignore the current load on edge nodes. Additionally, in the greedy version, the *LookAheadLoad* procedure is deactivated, and the weights α and β are set to 1 and 0, respectively.

Figure 10 illustrates the results of our multifaceted analysis. From a latency standpoint, the greedy algorithm is able to find good offloading candidates for mobile devices. As a matter of fact, the difference in terms of median latency achieved by greedy compared to the standard version of Nimbus is minimal. However, the standard deviation is

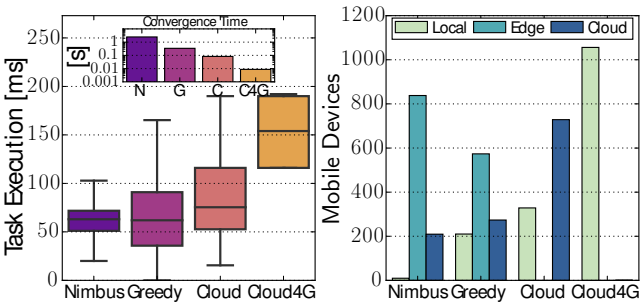


Fig. 10: Baseline comparison.

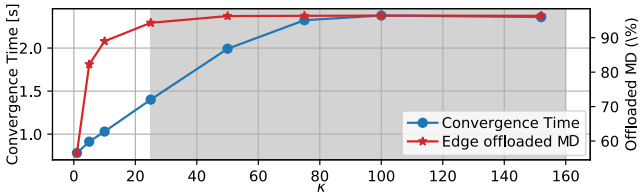


Fig. 11: Exploration Tradeoff (152 ENs, ∼1000 MDs).

much more noticeable due to the increasing number of non-offloaded MDs. Nimbus offloads ∼30% more MDs than the greedy version and, specifically, minimizes MDs that resort to using local resources for task execution. Note that these results are drawn over a largely homogeneous edge infrastructure, with only three classes of participating ENs. In a highly heterogeneous environment, the limited search scope used in the greedy configuration could lead to unstable results, since there is an increased chance of missing good offloading targets in the search procedure. We extract the cloud network RTTs from the RIPE Atlas dataset discussed in § 6). We obtained the network RTTs using probes pinging datacenters co-located in the same region. Additionally, we set the inference latency in the cloud to 5 ms (comparable to a T3-EN) regardless of the served devices (*e.g.*, no queue time). The *cloud-only* approach (labeled *Cloud* in Figure 10) produces acceptable results, but at the cost of slight higher median task latency and greater variance compared to Nimbus. Additionally, the approach is unable to offload tasks from many MDs, forcing them to run locally. Finally, the cloud-only variant with *mobile access network* (labeled *Cloud4G*) delivers the worst performance – with close to 100% MDs unable to offload their computation. The primary reasons are significantly expensive transmission and energy costs, and higher network RTT to the processing server. Our result is in line with previous research, which shows that mobile connections require significantly more energy per bit in transmission compared to Wi-Fi [43].

Figure 11 shows the relationship between percentage of edge-offloaded MD, convergence time of the algorithm, and value of κ for an edge-cloud infrastructure of 152 ENs and 1000 MDs. Regarding the algorithm convergence time, the greedy version performs one order of magnitude faster compared to the standard one (inset plot in Figure 10). It should be noted that the Nimbus cloud-only variants converge much faster due to their simplified solver logic. When in need to allocate high densities of MDs, properly tuning the exploration parameter κ allows us to find a convenient tradeoff between offloaded MDs ratio and algorithm convergence time. Selecting a value of 10 for κ allows to already offload ∼91% of the MD while keeping a sub-second convergence time. During our experiments with different infrastructure and ENs configurations, we noticed that setting κ between 10-20% of the total EN in the network strikes a good balance between MDs allocation percentage and convergence time. However, this cutoff point might also be affected by the rather strong homogeneity of our infrastructure, since we only consider three classes of ENs. We hypothesize that with a more heterogeneous network, the cutoff point would be higher which translates into a greater range of exploration and an increased cost in terms of convergence time.

Note that the convergence time in Figure 11 represents the time required to allocate all the MDs in the batch. The allocation operation does not run for every offloaded frame, but only once when the mobile devices initiate an offload request to the infrastructure. Additionally, the system is designed in such a way that, while the MD waits be offloaded, the end-user will not experience any service interruptions as the task will keep running locally until the allocation on the edge-cloud infrastructure is completed. In this case, we assume that the MD is capable of executing the task locally. Finally, it is valid to assume that the amount of time the user will spend using the infrastructure offsets greatly the allocation waiting time similarly to start-up latency in video streaming.

Takeaway 2. The greedy algorithm is able to find good offloading candidates for MDs faster than Nimbus at the cost of sub-optimal utilization of the edge-cloud resources (e.g., skipping good offloading targets in the search procedure). The cloud-only variant is effective but provides higher median task latency, increased energy consumption for the MD, and greater variance compared to Nimbus.

(D) Nimbus Variants. We developed three versions of our solver. The one used in the previous benchmarks was single-threaded (ST), meaning that the decision process was handled by a single controller node which had complete knowledge of the edge infrastructure. From a practical viewpoint, such a solver offers limited scalability, especially when both the size of the edge infrastructure and density of participating MDs increases. For such cases, the convergence time of the single-threaded variant becomes prohibitive. Therefore, we developed a multi-threaded (MT) variant of Nimbus, termed *MT Nimbus*, that makes it deployable in a distributed fashion. We applied a partitioning procedure to the edge-cloud infrastructure. For a simple-yet-effective solution, we adopted a naive approach where we created non-overlapping sets of ENs so that every thread (or, equivalently, the entity managing a network slice) is independent of the others. We are aware that the procedure followed to split the edge-cloud network resources is not optimal, but, in this context, it suffices the evaluation purpose. Transforming an algorithm from centralized to distributed entails additional costs as synchronizing different entities increases communication overheads. Our goal is to demonstrate the possibility of transforming our algorithm into a distributed form

and characterize its performance. In this work, we do not delve into communication and cross-node synchronization challenges of a distributed system and leave it for future work.

As the number of ENs for each tier can be non-proportional to the number of threads, the network slices created can be unbalanced. For example, with ten solver threads and three T3-EN, the first three threads would have in their network slice at most one T3-EN. While the principal benefit for our distributed algorithm is decreased convergence time, we sacrifice in *quality* of the solution as the algorithm is now less capable of fully exploiting the available edge-cloud infrastructure resources. Figure 12 shows the convergence time and task execution latency with an increasing number of threads. It can be observed that the more we slice the network, the fewer MDs are offloaded because each slice becomes *shallower*, thus reducing the degrees of exploration for the algorithm. However, the convergence time per-thread reduces by up to ∼**15**× when Nimbus uses four threads instead of one.

To mitigate the inefficient use of the edge-cloud infrastructure, we developed a *two-stage* solver version of Nimbus. In this variant, all the MDs not offloaded in the first distributed stage are scheduled for a second allocation pass. The second stage executes centrally and is modified so that it attempts to allocate the remaining MDs on the entire edge-cloud infrastructure (updated with the current load). This final variant is called *2PMT Nimbus* and the results obtained are shown in Figure 13.

While there is an additional cost in terms of convergence time due to the presence of a final aggregation step, the amount of non-offloaded MDs reduces drastically, especially with an increasing number of threads. The effective ratio of offloaded users also increases compared to MT-Nimbus as 2PMT-Nimbus tends to fit more MDs into the edge-cloud infrastructure. Overall, 2PMT-Nimbus does not violate any of the inherent constraints and is able to deliver the required quality of experience (e.g., FPS) to all the offloaded users. With only two threads, 2PMT-Nimbus achieves similar MD allocation ratios as the single-threaded version while almost halving the convergence time. With eight threads, 2PMT-Nimbus converges almost 3× faster than two-threads and offloads the majority of the users. While the convergence time achieved by 2PMT-Nimbus is much slower than MT-Nimbus, the former is able to allocate many more MDs at the edge-cloud infrastructure.

Note the anomaly in convergence time trend of 2PMT-Nimbus – where the convergence time increases despite an increased degree of parallelism. We explain the exception as follows. By assigning more threads, the generated network slices become shallower and fewer EN candidates are available to allocate MDs. The fewer users are allocated, the more effort is required by the centralized solver to complete the final reallocation step. This entails that the law of diminishing returns applies to the threads parallelism. In fact, with ten threads, the multi-threaded convergence time decreases, but the single-threaded increases. However, the overall performance in terms of allocation ratio looks better with increasing thread count. Consequently, if we would progressively increase the assigned threads boundlessly, we would circle back to the single-threaded performance, both for allocation and convergence time.

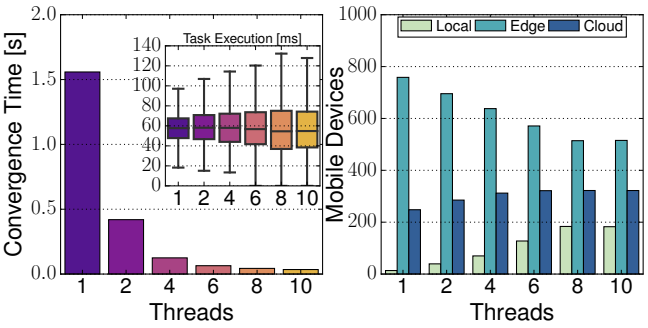


Fig. 12: Performance of MT-Nimbus.

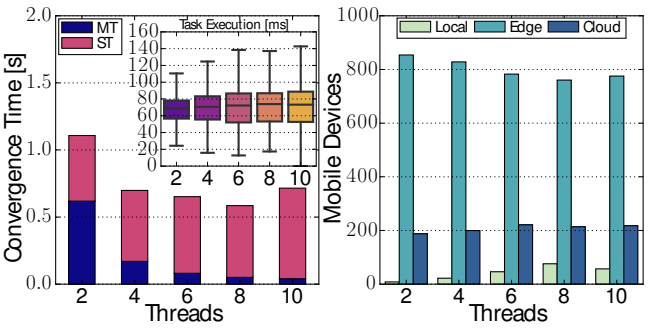


Fig. 13: Performance of 2PMT-Nimbus.

Takeaway 3. The ST version of Nimbus scales poorly as the size of the edge infrastructure and density of participating MDs increases. The MT variant is much faster but cannot fully makes use of the edge-cloud infrastructure. Finally, 2PMT-Nimbus provides the best performance in terms of ratio of offloaded MDs.

8 Limitations and Outlook

Edge computing will play a significant role in reshaping the future of cloud networks infrastructure. New applications and services will leverage information and processing capabilities offered at the network edge for varying purposes – including but not limited to data aggregation and analysis, multimedia content delivery, machine learning and AI. In this section, we explore orthogonal problems affecting edge computing putting our findings into a broader perspective.

Application & Network. Immersive applications, such as AR/VR, necessitate the deployment of edge servers in the network due to the strict latency constraints they impose. Such applications are guided by the human vestibular system which requires sensory inputs and interactions to be in complete sync; failure of which results in motion sickness and dizziness. As QoS of network communication technologies (*e.g.*, 5G and millimeter waves) improve (*i.e.*, shorter network delay and higher throughput [73, 76, 86]), optimizations in compute capabilities and task allocation mechanisms at edge become paramount to support multimedia QoE requirements.

However, end-to-end application latency still accounts for the most significant fraction of the perceived user experience, as discussed in § 7. Therefore, in this work, we focused on task execution time and network latency while ignoring the non-marginal overhead introduced by other components. These additional delays may have multiple sources, including the operating system, bloated network queues, network fluctuations (retransmissions, packet loss), to name a few. We ignore these variables in this work to keep the problem tractable since added delay caused by some of the above is predictable only to a certain extent. Consequently, our results should be considered as an optimistic estimate on top of which application logic and context overhead must be added. The Nimbus system presented in this paper manages the interaction between edge infrastructure and MDs and offers a device-independent framework to offload tasks to the edge. In our future work, we plan to extend the platform to calculate the additional application overheads, as discussed above.

Smartphone evolution. The symbiosis between edge computing and mobile-based applications is complicated. Factors like ever-increasing computational capacities of smartphones [12, 13], and more general-purpose utility of edge computing begs re-thinking the applicability of edge for mobile clients. For example, high-end smartphones equipped with powerful mobile GPUs benefit more from running computations locally than offloading, due to higher efficiency (in energy consumption and inference time) offered by their processor architectures and algorithms [15]. On the other hand, essential operations utilizing local GPU may become throttled as number of applications competing for the shared GPU cycles increases. We feel that edge resources can be used to further enhance (or enable) what can be achieved by a smartphone. An example could be executing more sophisticated and accurate neural networks – which are often prohibitive for smartphones as they require considerably more RAM and computational power. Until mobile devices are battery-powered, there will always be a trade-off in performance versus battery consumption. One can also envision smartphones becoming part of the edge infrastructure [48], which poses new and exciting challenges for managing transient, mobility capable compute nodes.

Security Implications. We purposefully avoid delving into possible security vulnerabilities of Nimbus since we consider it out-of-scope. Here, we explore possible security holes in our system and provide hints on how to mitigate them. In our approach, we do not restrict an MD to the maximum time for which they can utilize the edge-cloud infrastructure. This can lead to numerous problems: a malicious MD might decide to offload tasks forever and to multiple servers to leech resources from the infrastructure, which may lead to starvation. One solution could be to use a credit or reputation system [54], where an MD can only utilize services offered by the edge-cloud by spending some virtual currency. Other possible approaches could be introducing a fixed time limit after which the MD is forcefully rescheduled. However, all these solutions require MD to be registered so that system can keep track of their credit or the amount of time spent using the service. Distributed ledgers and blockchain might be useful in this scenario to help keep track of the user credit and enable point-to-point payments [44, 70].

In § 7, we discussed the possibility of an *elastic* infrastructure composed of *consumer* ENs offering compute resources similarly to [18] to respond to network overloads. There are several issues associated with such an infrastructure, including reduced control over ENs, intermittent resource availability, reliability, inconsistent execution and queue time predictions, and security and privacy concerns. Additionally, *trust* can be a problem for such an infrastructure as malicious ENs might extract sensitive information while computing a task or deliberately modify the outcome to disrupt the service. Possible resolutions could be employing Trust Execution Environments (TEE) [69, 75] to secure the compute steps at the cost of operational complexity.

Deployment Challenges. When discussing changes advocated by edge computing, it is essential to keep in mind its *adoption* cost. Depending on the type of deployed ENs, the Capital Expenditures (CapEx) [2] and Operational Expenditures (OpEx) cost demand careful planning of the infrastructure as function of the QoS to be delivered over a period of time. Similar to cloud and ISP services, edge-cloud could employ a subscription-based operation model. End-users could choose from different subscription plans that best cater to desired QoE of targeted applications, e.g. gaming, healthcare, video analytics, etc.

9 Conclusion

In this paper we presented Nimbus, a multi-objective task allocation solution that can minimize the latency of mobile real-time object detection models by offloading them to an edge-cloud infrastructure. Based on an extensive set of real data and measurements, our multifaceted evaluation benchmarks three ever-improving variants of Nimbus addressing, especially, the problem of scalability from the infrastructure and end-users point of view. We verify the effectiveness of Nimbus through trace-driven simulations. Based on an extensive set of real data and measurements, we show the potential of Nimbus in boosting the performance of AR applications when offloaded from mobile devices to an edge-cloud infrastructure. Additionally, our multifaceted evaluation presents three ever-improving variants of Nimbus addressing, especially, scalability issues of edge-cloud infrastructure. Finally, in light of our algorithm and approach, we discuss several crucial open questions concerning edge computing and highlights future research directions.

References

[1] Benchmarking Hardware for CNN Inference in 2018. https://towardsdatascience.com/benchmarking-hardware-for-cnn-inference-in-2018-1d58268de12a.

[2] Capital Expenditure. https://en.wikipedia.org/wiki/Capital_expenditure. Accessed: 2020-04-27.

[3] Hill Climbing. https://en.wikipedia.org/wiki/Hill_climbing. Accessed: 2020-04-27.

[4] Network Latency Datasets. https://github.com/uofa-rzhu3/NetLatency-Data. Accessed: 2020-06-30.

[5] Leibniz-Rechenzentrum. https://www.lrz.de, . Accessed: 2020-05-22.

[6] WLAN-Verbindungen im MWN (Statistik). http://wlan.lrz.de/apstat/search, . Accessed: 2020-05-22.

[7] How is Mobile AR Landing with Consumers? https://virtualrealitypop.com/how-is-mobile-ar-landing-with-consumers-cbc4b14e5957. Accessed: 2020-04-27.

[8] PlanetLab. https://www.planet-lab.org. Accessed: 2020-06-30.

[9] A First Inside Look at Pokémon GO Battery Drain. http://mobileenerlytics.com/a-first-inside-look-at-pokemon-go-battery-drain-you-wont-catch-many-if-your-battery-dies-so-quickly/. Accessed: 2020-04-02.

[10] Latency – the sine qua non of AR and VR. http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/. Accessed: 2020-04-27.

[11] Seattle. https://seattle.poly.edu. Accessed: 2020-06-30.

[12] Your Phone Is Now More Powerful Than Your PC. https://insights.samsung.com/2018/08/09/your-phone-is-now-more-powerful-than-your-pc/, . Accessed: 2020-05-22.

[13] How the computing power in a smartphone compares to supercomputers past and present. https://www.businessinsider.com/infographic-how-computing-power-has-changed-over-time-2017-11?r=DE&IR=T, . Accessed: 2020-05-22.

[14] NVIDIA Triton Inference Server. https://github.com/NVIDIA/triton-inference-server. Accessed: 2020-05-22.

[15] UbiSpark project. https://ubispark.cs.helsinki.fi/. Accessed: 2020-06-30.

[16] Mario Almeida, Stefanos Laskaridis, Stylianos I Venieris, Ilias Leontiadis, and Nicholas D Lane. Dyno: Dynamic onloading of deep neural networks from cloud to device. *arXiv preprint arXiv:2104.09949*, 2021.

[17] Brandon Amos, Bartosz Ludwiczuk, Mahadev Satyanarayanan, et al. Openface: A general-purpose face recognition library with mobile applications. *CMU School of Computer Science*, 6:2, 2016.

[18] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[19] Mykhaylo Andriluka, Stefan Roth, and Bernt Schiele. People-tracking-by-detection and people-detection-by-tracking. In *2008 IEEE Conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2008.

[20] Mohammad Babar, Muhammad Sohail Khan, Farman Ali, Muhammad Imran, and Muhammad Shoaib. Cloudlet computing: Recent advances, taxonomy, and challenges. *IEEE Access*, 9:29609–29622, 2021.

[21] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 272–285, 2007.

[22] Paul E Black. Greedy algorithm. *Dictionary of Algorithms and Data Structures*, 2:62, 2005.

[23] Michaela Blott, Lisa Halder, Miriam Leeser, and Linda Doyle. Qutibench: Benchmarking neural networks on heterogeneous hardware. *J. Emerg. Technol. Comput. Syst.*, 15(4), dec 2019. ISSN 1550-4832. doi: 10.1145/3358700. URL https://doi.org/10.1145/3358700.

[24] Mathieu Bouet and Vania Conan. Mobile edge computing resources optimization: A geo-clustering approach. *IEEE Transactions on Network and Service Management*, 15 (2):787–796, 2018.

[25] Tristan Braud, Farshid Hassani Bijarbooneh, Dimitris Chatzopoulos, and Pan Hui. Future networking challenges: The case of mobile augmented reality. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1796–1807. IEEE, 2017.

[26] Tristan Braud, Pengyuan Zhou, Jussi Kangasharju, and Pan Hui. Multipath computation offloading for mobile augmented reality. In *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–10. IEEE, 2020.

[27] Martin Breitbach, Janick Edinger, Siim Kaupmees, Heiko Trötsch, Christian Krupitzer, and Christian Becker. Voltaire: Precise energy-aware code offloading decisions with machine learning. In *2021 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–10. IEEE, 2021.

[28] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E. Culler, and Randy H. Katz. Marvel: Enabling mobile augmented reality with low energy and low latency. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, page 292–304, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3274783.3274834.

[29] Ying Chen, Ning Zhang, Yongchao Zhang, and Xin Chen. Dynamic computation offloading in edge computing for internet of things. *IEEE Internet of Things Journal*, 6(3):4242–4251, 2018.

[30] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314, 2011.

[31] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. Surrounded by the Clouds: A Comprehensive Cloud Reachability Study. In *Proceedings of The Web Conference 2021*, WWW '21, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3442381.3449854. URL https://doi.org/10.1145/3442381.3449854.

[32] Lorenzo Corneo, Nitinder Mohan, Aleksandr Zavodovski, Walter Wong, Christian Rohner, Per Gunningberg, and Jussi Kangasharju. (how much) can edge computing change network latency? In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2021.

[33] V. Cozzolino, J. Ott, A. Y. Ding, and R. Mortier. Ecco: Edge-cloud chaining and orchestration framework for road context assessment. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 223–230, 2020.

[34] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international*

conference on Mobile systems, applications, and services, pages 49–62, 2010.

[35] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 121–135, 2015.

[36] The Khang Dang, Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Jörg Ott, and Jussi Kangasharju. Cloudy with a Chance of Short RTTs: Analyzing Cloud Connectivity in the Internet. In *Proceedings of Internet Measurement Conference*, IMC '21, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3487552.3487854. URL https://doi.org/10.1145/3487552.3487854.

[37] Kalyanmoy Deb. Multi-objective optimization. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 403–449. Springer, 2014.

[38] Thinh Quang Dinh, Jianhua Tang, Quang Duy La, and Tony QS Quek. Offloading in mobile edge computing: Task allocation and computational frequency scaling. *IEEE Transactions on Communications*, 65(8):3571–3584, 2017.

[39] Utsav Drolia, Rolando Martins, Jiaqi Tan, Ankit Chheda, Monil Sanghavi, Rajeev Gandhi, and Priya Narasimhan. The case for mobile edge-clouds. In *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*, pages 209–215. IEEE, 2013.

[40] Maximilian Eder, Lorenzo Corneo, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. Surrounded by the clouds, 2021. URL https://mediatum.ub.tum.de/1593899.

[41] Jason Flinn and Z Morley Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 84–89, 2011.

[42] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. {COMET}: Code offload by migrating execution transparently. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 93–106, 2012.

[43] Tian Guo. Cloud-based or on-device: An empirical study of mobile deep inference. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 184–190. IEEE, 2018.

[44] Ye Guo and Chen Liang. Blockchain application and outlook in the banking industry. *Financial Innovation*, 2 (1):24, 2016.

[45] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016.

[46] Zhenhua Han, Haisheng Tan, Guihai Chen, Rui Wang, Yifan Chen, and Francis CM Lau. Dynamic virtual machine management via approximate markov decision process. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[47] Ting He, Hana Khamfroush, Shiqiang Wang, Tom La Porta, and Sebastian Stein. It's hard to share: Joint service placement and request scheduling in edge clouds with sharable and non-sharable resources. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 365–375. IEEE, 2018.

[48] Junyan Hu, Kenli Li, Chubo Liu, and Keqin Li. Game-based task offloading of multiple mobile devices with qos in mobile edge computing systems of limited computation capacity. *ACM Trans. Embed. Comput. Syst.*, 19(4), July 2020. ISSN 1539-9087. doi: 10.1145/3398038. URL https://doi.org/10.1145/3398038.

[49] C-L Hwang and Abu Syed Md Masud. *Multiple objective decision making—methods and applications: a state-of-the-art survey*, volume 164. Springer Science & Business Media, 2012.

[50] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European conference on computer vision (ECCV)*, pages 0–0, 2018.

[51] Mike Jia, Jiannong Cao, and Weifa Liang. Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks. *IEEE Transactions on Cloud Computing*, 5(4):725–737, 2015.

[52] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[53] Kuljeet Kaur, Tanya Dhand, Neeraj Kumar, and Sherali Zeadally. Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE wireless communications*, 24(3): 48–56, 2017.

[54] Michael Kinateder and Kurt Rothermel. Architecture and algorithms for a distributed reputation system. In *International Conference on Trust Management*, pages 1–16. Springer, 2003.

[55] Chenge Li, Gregory Dobler, Xin Feng, and Yao Wang. Tracknet: Simultaneous object detection and tracking and its application in traffic video analysis. *arXiv preprint arXiv:1902.01466*, 2019.

[56] Fangming Liu, Peng Shu, Hai Jin, Linjie Ding, Jie Yu, Di Niu, and Bo Li. Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications. *IEEE Wireless communications*, 20(3):14–22, 2013.

[57] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[58] Qiang Liu, Siqi Huang, Johnson Opadere, and Tao Han. An edge network orchestrator for mobile augmented

reality. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 756–764. IEEE, 2018.

[59] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312. IEEE, 2014.

[60] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[61] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 286–299. IEEE, 2018.

[62] Jiaying Meng, Wenbin Shi, Haisheng Tan, and Xiangyang Li. Cloudlet placement and minimum-delay routing in cloudlet computing. In *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, pages 297–304. IEEE, 2017.

[63] Jiaying Meng, Haisheng Tan, Chao Xu, Wanli Cao, Liuyan Liu, and Bojie Li. Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2287–2295. IEEE, 2019.

[64] Nitinder Mohan and Jussi Kangasharju. Edge-fog cloud: A distributed cloud for internet of things computations. In *2016 Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2016.

[65] Nitinder Mohan and Jussi Kangasharju. Placing it right!: optimizing energy, processing, and transport in edge-fog clouds. *Annals of Telecommunications*, 73(7-8):463–474, 2018.

[66] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. Pruning edge research with latency shears. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 182–189, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381451. doi: 10.1145/3422604.3425943. URL https://doi.org/10.1145/3422604.3425943.

[67] Roberto Morabito, Vittorio Cozzolino, Aaron Yi Ding, Nicklas Beijar, and Jorg Ott. Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111, 2018.

[68] Ashkan Nikravesh, David R Choffnes, Ethan Katz-Bassett, Z Morley Mao, and Matt Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *International Conference on Passive and Active Network Measurement*, pages 12–22. Springer, 2014.

[69] Zhenyu Ning, Jinghui Liao, Fengwei Zhang, and Weisong Shi. Preliminary study of trusted execution environments on heterogeneous edge platforms. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 421–426. IEEE, 2018.

[70] Jianli Pan, Jianyu Wang, Austin Hester, Ismail Alqerm, Yuanni Liu, and Ying Zhao. Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts. *IEEE Internet of Things Journal*, 6(3): 4719–4732, 2018.

[71] Horst Possegger, Thomas Mauthner, Peter M Roth, and Horst Bischof. Occlusion geodesics for online multi-object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1306–1313, 2014.

[72] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.

[73] Theodore S Rappaport, Shu Sun, Rimma Mayzus, Hang Zhao, Yaniv Azar, Kevin Wang, George N Wong, Jocelyn K Schulz, Mathew Samimi, and Felix Gutierrez. Millimeter wave mobile communications for 5g cellular: It will work! *IEEE access*, 1:335–349, 2013.

[74] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[75] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[76] J. Sachs, G. Wikstrom, T. Dudda, R. Baldemair, and K. Kittichokechai. 5g radio network design for ultra-reliable low-latency communication. *IEEE Network*, 32 (2):24–31, 2018.

[77] Onur Sahin and Ayse K Coskun. Providing sustainable performance in thermally constrained mobile devices. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 72–77, 2016.

[78] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[79] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4): 14–23, 2009.

[80] Ryan Shea, Di Fu, and Jiangchuan Liu. Rhizome: Utilizing the public cloud to provide 3d gaming infrastructure. In *Proceedings of the 6th ACM Multimedia Systems Conference*, pages 97–100, 2015.

[81] Shu Shi and Cheng-Hsin Hsu. A survey of interactive remote rendering systems. *ACM Computing Surveys (CSUR)*, 47(4):1–29, 2015.

[82] RIPE NCC Staff. Ripe atlas: A global internet measurement network. *Internet Protocol Journal*, 18(3), 2015.

[83] Xiang Sun and Nirwan Ansari. Edgeiot: Mobile edge computing for the internet of things. *IEEE Communications Magazine*, 54(12):22–29, 2016.

[84] Haisheng Tan, Zhenhua Han, Xiang-Yang Li, and Francis CM Lau. Online job dispatching and scheduling in edge-clouds. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[85] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[86] R. Trivisonno, R. Guerzoni, I. Vaishnavi, and D. Soldani. Towards zero latency software defined 5g networks. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 2566–2571, 2015.

[87] Blesson Varghese, Eyal De Lara, Aaron Yi Ding, Cheol-Ho Hong, Flavio Bonomi, Schahram Dustdar, Paul Harvey, Peter Hewkin, Weisong Shi, Mark Thiele, et al. Revisiting the arguments for edge computing research. *IEEE Internet Computing*, 25(5):36–42, 2021.

[88] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016.

[89] Fangxin Wang, Miao Zhang, Xiangxiang Wang, Xiaoqiang Ma, and Jiangchuan Liu. Deep learning for edge computing applications: A state-of-the-art survey. *IEEE Access*, 8:58322–58336, 2020.

[90] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 63–71. IEEE, 2018.

[91] X. Wang, L. T. Yang, X. Xie, J. Jin, and M. J. Deen. A cloud-edge computing framework for cyber-physical-social services. *IEEE Communications Magazine*, 55(11): 80–85, 2017.

[92] Yue Wang, Xiaofeng Tao, Xuefei Zhang, Ping Zhang, and Y Thomas Hou. Cooperative task offloading in three-tier mobile computing networks: An admm framework. *IEEE Transactions on Vehicular Technology*, 68(3):2763–2776, 2019.

[93] Jiyan Wu, Chau Yuen, Ngai-Man Cheung, Junliang Chen, and Chang Wen Chen. Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12):1988–2001, 2015.

[94] Chunwei Xia, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. Dnntune: Automatic benchmarking dnn models for mobile-cloud computing. *ACM Trans. Archit. Code Optim.*, 16(4), December 2019. ISSN 1544-3566. doi: 10.1145/3368305. URL https://doi.org/10.1145/3368305.

[95] Zhujun Xiao, Zhengxu Xia, Haitao Zheng, Ben Y Zhao, and Junchen Jiang. Towards performance clarity of edge video analytics. *arXiv preprint arXiv:2105.08694*, 2021.

[96] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 15. ACM, 2017.

[97] Wenxiao Zhang, Bo Han, and Pan Hui. On the networking challenges of mobile augmented reality. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pages 24–29, 2017.

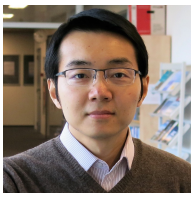
[98] Junlong Zhou, Tian Wang, Peijin Cong, Pingping Lu, Tongquan Wei, and Mingsong Chen. Cost and makespan-aware workflow scheduling in hybrid clouds. *Journal of Systems Architecture*, 100:101631, 2019.

[99] Agustin Zuniga, Huber Flores, Eemil Lagerspetz, Petteri Nurmi, Sasu Tarkoma, Pan Hui, and Jukka Manner. Tortoise or hare? quantifying the effects of performance on mobile app retention. In *The World Wide Web Conference*, pages 2517–2528, 2019.

Vittorio Cozzolino is a PhD candidate in the Chair of Connected Mobility at the Technical University of Munich, and his research interests are in distributed systems, edge and cloud computing, resource management and virtualization.

Leonardo Tonetto is a PhD candidate in the Chair of Connected Mobility at the Technical University of Munich, and his research interests are in mobile user behavioral modeling, human mobility and complex networks.

Dr. Nitinder Mohan is a Postdoctoral researcher in the Chair of Connected Mobility at Technical University of Munich, Germany. He received his Ph.D. (as Marie Curie ITN fellow) from the Department of Computer Science at the University of Helsinki in Finland and M.Tech. degree (honors) from Indraprastha Institute of Information Technology Delhi (IIIT-D), India. He has been awarded "Outstanding Ph.D. Dissertation Award" by IEEE Technical Committee on Scalable Computing (TCSC). He has worked as Visting Researcher in NEC Labs Europe, Germany and at University of Göttingen, Germany, and as a Project Scientist in Indian Institute of Technology Delhi (IIT-D), India.



Aaron Yi Ding leads the Cyber-Physical Intelligence Lab as a Tenured Faculty of TU Delft and Associate Professor (Docent) in Computer Science at the University of Helsinki. His research focuses on Edge AI solutions for cyber-physical systems in smart health, mobility and energy domains. He is an associate editor for ACM Transactions on Internet of Things (TIOT), IEEE OJ-ITS and Multimodal Technologies and Interaction. He received his MSc and PhD degrees from the Department of Computer Science (Birthplace of Linux) at the University of Helsinki. With over 14 years of R&D experience across EU, UK and USA, he has worked at TU Munich with Joerg Ott, at University of Cambridge with Jon Crowcroft and at Columbia University with Henning Schulzrinne. He has 60+ peer reviewed publications, receiving best paper awards and recognition from ACM SIGCOMM, ACM EdgeSys, ACM SenSys CCIoT, IEEE INFOCOM, and the esteemed Nokia Foundation Scholarships.

Jörg Ott holds the Chair of Connected Mobility at Technische Universität München in the Faculty of Informatics since August 2015. He holds a PhD (1997) and a diploma in Computer Science (1991) from TU Berlin and a diploma in Industrial Engineering from TFH Berlin (1995). His research interests are in network architectures,protocols, and algorithms for connecting mobile nodes to the Internet and to each other. He explores edge and in-network computing as well as decentralized services, with a particular interest in privacy.

Publication V

Cozzolino, V., Moroz, O., & Ding, A. Y. (2018). The Virtual Factory: Hologram-Enabled Control and Monitoring of Industrial IoT Devices. In Proceedings of 2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR 2018) (pp. 120-123). Institute of Electrical and Electronics Engineers (IEEE).https://doi.org/10.1109/AIVR.2018.00024

Publication Summary

The integration of smart devices in domestic, industrial and commercial environments has profoundly reshaped the way we interact with our surrounding. Specifically within industry, Internet of Things (IoT) is currently adopted to solve multiple problems such as smart labeling, energy management, control and monitoring , demonstrating the constructive uses of digitalization and smart automation. As machines and industrial physical processes change, the interfaces to interact with them should also change. Until a few years ago, fixing or tuning machines in a factory required manual intervention. To build the interaction between the physical and virtual worlds, we promote the exploration of new interactive experiences via augmented reality (AR). In this work, we focus on understanding and evaluating the extent to which AR can help to interact with complex machines through direct, visual, three-dimensional (3D) feedback.

For this work, our contributions can be summarized as follows: (i) design and implementation of a flexible AR platform where new IoT devices can be easily plugged-in and integrated into the virtual factory workflow, (ii) a user study to determine the effectiveness of AR based interaction versus classic SCADA-like systems, and (iii) insight from the performance evaluation that reveals the limitations of existing HMD devices that deserve future research from the community.

Our prototype implementation was the mean through which we conducted the user-study aimed at evaluating holographic interfaces compared against traditional interfaces. Our study showed that the majority of the participants found holographic manipulation more attractive and natural to interact with. However, what emerged was that current performance characteristics of head-mounted displays must be improved to be applied in production.

Author's Contribution

I came up with the idea for the paper as a foundation for the integration of IoT devices with augmented reality (AR) interfaces. I contributed to the system design aspects and research foundation while Oleksii Moroz implemented and evaluated the system.

The Virtual Factory: Hologram-Enabled Control and Monitoring of Industrial IoT Devices

Vittorio Cozzolino
Technical University of Munich
Munich, Germany
cozzolin@tum.de

Oleksii Moroz
Soley GmbH
Munich, Germany
oleksii.moroz@hotmail.com

Aaron Yi Ding
Delft University of Technology
Delft, Netherlands
aaron.ding@tudelft.nl

***Abstract*—Augmented reality (AR) has been exploited in manifold fields but is yet to be used at its full potential. With the massive diffusion of smart devices, opportunities to build immersive human-computer interfaces are continually expanding. In this study, we conceptualize a virtual factory: an interactive, dynamic, holographic abstraction of the physical machines deployed in a factory. Through our prototype implementation, we conducted a user-study driven evaluation of holographic interfaces compared to traditional interfaces, highlighting its pros and cons. Our study shows that the majority of the participants found holographic manipulation more attractive and natural to interact with. However, current performance characteristics of head-mounted displays must be improved to be applied in production.**

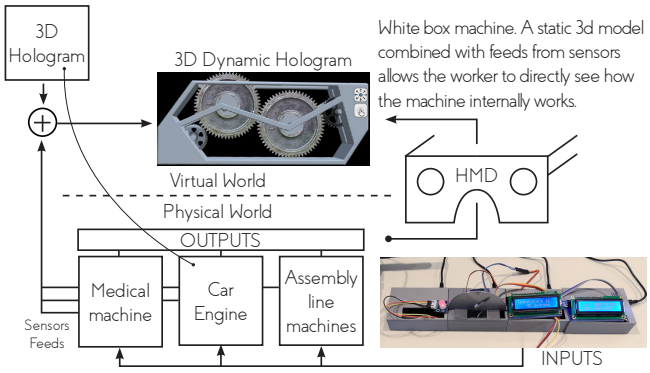


Fig. 1. Bridging Physical and Virtual World

I. Introduction

The integration of smart devices in domestic, industrial and commercial environments has profoundly reshaped the way we interact with our surrounding. Specifically within industry, Internet of Things (IoT) is currently adopted to solve multiple problems as smart labeling [1], energy management, control and monitoring [2], demonstrating the constructive uses of digitalization and smart automation.

As machines and industrial physical processes change, the interfaces to interact with them should also change. Until a few years ago, fixing or tuning machines in a factory required manual intervention. Today, most information about the state of physical processes is collected using Supervisory control and data acquisition (SCADA) systems and monitored by human operators. Particularly, virtual sensors are already explored in the industry to visualize exact simulations of motors [3]. To strengthen the relation between the physical and virtual worlds, we promote the exploration of new interactive experiences via AR. Our primary concern is to understand and evaluate the extent to which AR can help to interact with complex machines through direct, visual, three-dimensional (3D) feedback (although this could easily be extended to other environments).

The wide-spread diffusion of portable head-mounted displays (HMD), such as HoloLens, Lenovo Explorer Headset, HTC Vive, and Oculus Rift, has opened doors to a new paradigm in which the physical world becomes the user interface. AR and virtual reality (VR) have been already utilized in diverse fields, such as tourism, navigation, education, information management. In each of these instantiations, the augmented interface is meant to provide auxiliary information about the surrounding environment to users, thereby helping them to complete specific tasks significantly faster and more accurately [4]. Narrowing our focus to a fabrication scenario, we aim to provide factory workers a more contextualized and visual representation of the real-time, evolving state of a complex machine in the virtual world (through holograms).

Fig.1 shows that with AR, the physical model is represented by inputs, outputs, and readings from sensors. The physical model becomes a virtual, dynamic model based on these parameters. Hence, a potential worker can actually *see* the way a machine works, given the availability of a 1:1 holographic model matching it. Such an interaction has another benefit: it simplifies the knowledge transfer from old to new employees. Half of the human brain is directly or indirectly devoted to processing visual information and visual feedback is represented in our brain into a spatio-temporal pattern of cerebral excitation [5]. Additionally, visual stimuli generates neural signals in the amygdala tying the brain reinforcement learning process to emotions, possibly enhancing the cognitive behaviour [6].

Both HMDs and latest smartphones can provide AR experiences. Herein, we focus on the importance of HMD as they offer hands-free interaction, which is a clear benefit when working in a factory (or any other work environ-

ment). However, AR glasses (or HMDs) are not supported by a majority of currently available AR solutions. Instead, most AR solution frameworks are aimed at hand-held devices or particular operating systems, such as Android or iOS [7]. Therefore, there is a need to bridge the gap and develop new AR applications especially for industrial environment where the use of hand-held devices is often not possible.

This study presents a prototype framework to enable users to interact with complex machinery and complete tasks via hologram-based interaction. By supplying a dynamic, 3D hologram that changes according to the interaction with nearby smart devices, we want to assess the benefit of providing visual *dynamic* representation on top of virtual information about the system (as in AR annotations [8]). The system comprises three main elements: smart sensors and actuators, the HoloLens HMD, and the Unity engine. In particular, our contributions are as follows:

- Design and implementation of a flexible AR platform where new IoT devices can be easily plugged-in and integrated into the virtual factory workflow.
- A user study to determine the effectiveness of AR-based interaction versus classic SCADA-like systems.
- Insight from the performance evaluation that reveals the limitations of existing HMD devices that deserve future research from the community.

II. System Design and Implementation

Fig.2 presents the three-layer architecture of the system. The IoT layer comprises the network of IoT devices, such as smart sensors and actuators, used to interact with the system. The end-user layer is the core of our system; it provides the holographic abstraction of the physical world. The edge layer is primarily responsible for storage, administration, and organization of the local network; main management operations are handled by this layer. For the initial prototype design, we modelled a complex machinery as an ensemble of embedded boards equipped with sensors and actuators. This design choice forces the users to change their position to interact with different physical controllers; thus, this design choice was particularly important in our user study.

A. IoT Layer

This layer handles the communication with sensors and actuators connected to different embedded devices. The available physical devices within range are a part of this layer and share their capabilities with the end-user layer via a simple web protocol. Hence, an initial setup phase is required to reveal the available sensors and actuators connected to the system and associate them with a machine. To do so, a semantic representation of the device functionality is exchanged with the end-user layer and used for automated build of holographic interface.

Implementation details. For this layer we used multiple embedded devices. The backend software to communicate with the other layers was developed by combining Python and Raspberry Pi. The GPIO library was used to handle sensors and actuators, and the flask framework was used for server management and interaction over the network.

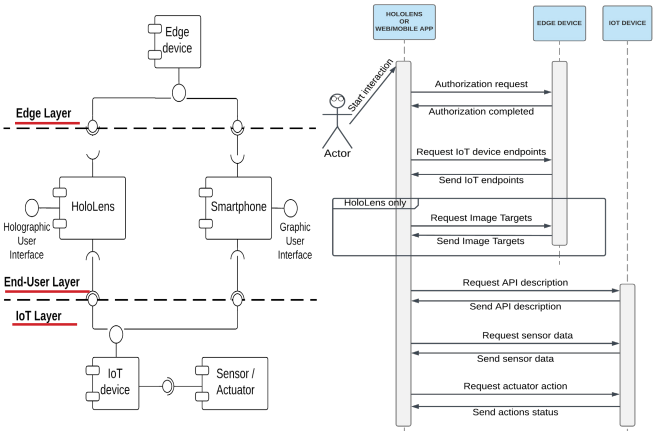


Fig. 2. System Representation

B. End-User Layer

The end-user layer is the core of our system and handles the organization and spatial recognition of the holograms. It is built as an event-driven application based on Unity and is composed of four main modules. The *UI Manager* is responsible for automated generation of holographic interfaces based their semantic representation. The *Event Manager* manages the information about IoT devices and processes device detection and interaction events to update the UI Manager. The *Server Manager* is the core communication module and is responsible for all data exchanges between system devices. In addition, the Server Manager loads the recognition models and semantic data from IoT devices. The *Semantic Module* is the data layer of the application and stores information about IoT devices and their virtual representation plus the specifics available functionalities.

Implementation details. The end-user layer was implemented in C# with Mixed Reality Toolkit libraries, and it runs directly on HoloLens. Object detection and tracking is implemented with the Vuforia AR SDK.

C. Edge Layer

The physical interaction between the headset and a machine happens only when the user is in direct proximity to the relevant IoT sensor; we decided to reflect this feature in our system design. In particular, instead of storing all the information regarding a group of smart devices on the cloud, we collected configuration and capabilities of the smart devices at the edge. Hence, to access the holographic interface of a specific machine, it is necessary to be in its proximity. This makes sense because the necessity of visualizing a 3D model of a physical objects arises only when we are close to it.

The deployed edge device is responsible for a cluster of IoT nodes in proximity: it stores IP endpoints and object recognition models of smart devices that are used by the end-user layer.

Implementation details. The backend application running in this layer to store information about the devices in the network was developed as a combination of Node.js and MongoDB.

III. Evaluation

We conducted two types of evaluation: user study and application benchmarking. For the experiments, multiple embedded boards equipped with sensors and actuators were installed in a room. Each actuator or sensors controlled a specific component of the machine (e.g., a spinning gear). Physical manipulation of these devices changed the state of specific components inside the 3D model of the machine. The system starts in an unstable state and the goal was to bring the machine to a stable state opportunely tuning different components (e.g. align spinning gears, control their speed, avoid overheating). Users were notified about the task completion through the interface they were using: either HMD and holograms or a SCADA-like web interface and a tablet. When using the HMD, the hologram changed in real-time according to the user inputs. In contrast, the web interface only provided textual feedback.

A. User Study

The usability test was aimed at answering two distinct questions: **Q1.** *How do participants receive the usage of the holographic technology?* and **Q2.** *How do holographic interfaces fare compared with standard ones?* The experiment was completed in four days and involved 22 participants (19 males and 3 females). Each user interacted with the system for 20 minutes. Most participants had a background in computer science and previous experience with AR or VR headsets. Only 18% of participants had previous experience with HoloLens.

During our usability test, the participants were asked to firstly get used to the HMD and the holograms technology and then evaluate the holographic interface interaction with our application. After the test, the users were asked to fill out a questionnaire. The questionnaire about design-oriented development was based on the study reported by Wich et al. on usability-evaluation questionnaires [9].

We observed that users feel uncertain about the convenience of holograms and were sceptical about the possibility of integration in their daily life. However, there is indeed a trend showing that holographic interfaces are in general more attractive as participants had a positive experience with the holographic manipulation. These results are summarized in Figure 3. New users grew accustomed to the holographic interface quickly and felt more confident after learning the basics.

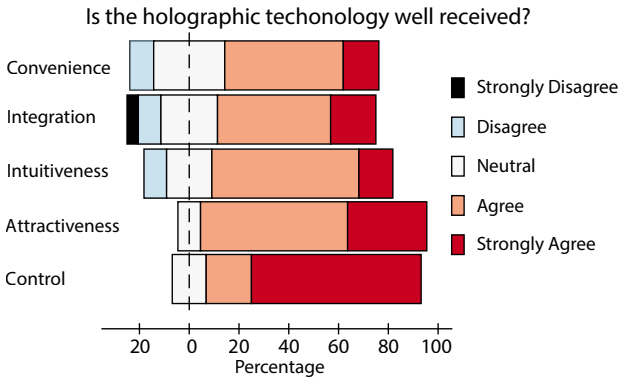


Fig. 3. The holograms technology is generally well received

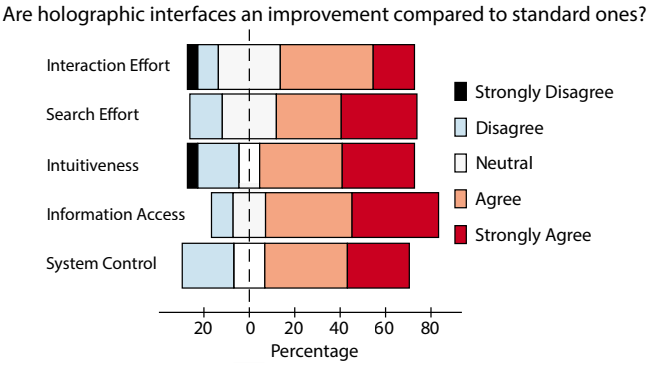


Fig. 4. Holographic interfaces proved to simplify manifold interactions compared to traditional ones

For the second part of the study, users were asked to compare their experiences of the two interfaces and express their preference. Result are shown in Figure 4. The majority of the participants preferred the holographic application (despite the manifold issues experienced with HoloLens) and stated that interacting with the SCADA-like web interface required greater effort. Only a small percentage of the participants expressed their scepticism regarding the holographic interface asserting to not feel confident during the interaction with holograms. The negative score of interaction effort and intuitiveness is related to the following concerns expressed by the users: inaccurate gesture recognition windows, narrow field of view, abrupt gaze pointer and headset weight and placement.

B. Performance Analysis

Figure 5 shows preliminary performance results of our application (average of 10 iterations) collected with Windows Performance Recorder and successively analysed with the Windows Performance Analyzer. The measurement granularity is one data-point/s. System power consumption represents the amount of power complexly used by Hololens while SoC power consumption amounts only for CPU, GPU and memory. All values (except FPS) are represented as percentage. Power consumption was

definitely high during all our experiments, the application posed a lot of stress particularly on the GPU leading to high SoC power consumption values. Considering the device's autonomy of 113 minutes and its charging time of 1 h, we conclude that either the battery should be optimized or developers must find a good trade-off between application functionalities and battery life. CPU utilization was reasonable with peaks caused by the Vuforia image recognition process, which includes the loading of recognition data and IoT components discovery. Thus, based on values of the processor load during the interaction, we conclude that the HoloLens has sufficient CPU power for image recognition tasks. GPU usage is heavily affected by the UI panel rendering, which also influences placement of and interaction with holograms. FPS were definitely acceptable with an average of 48 and the usability testing showed that even with just 20 FPS (during complex holographic visualizations) the user experience was not compromised.

In our tests, we assessed that the Hololens can overheat. We used a ThermalSeek IR camera to monitor the device temperature over time. After an average of 30 minutes, it reached a peak of 43.3° Celsius (our lab temperature was 29° Celsius) and constantly switched to a *cooldown* state effectively preventing any kind of interaction. Such behaviour breaks the user experience and allegedly render the device not designed for prolonged utilization.

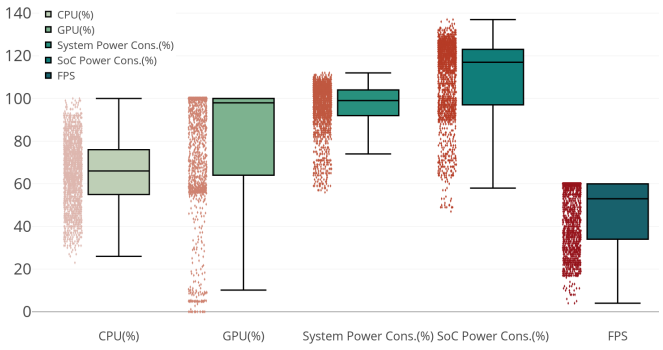


Fig. 5. Holographic application performance

IV. Related Work

There have been multiple attempts to integrate AR with smart devices. In fact, augmented reality was recently announced as one of ideal interfaces in IoT; its layer offers an abstraction that provides a simplified view on smart things and hides all irrelevant technical details from users [10]. Factory of the Future [11] describes factories as the perfect use case for the IoT object manipulation through augmented reality. It introduces a multi-modal and multi-client system for a huge factory which supports workers on their workplaces and provides a control interface through augmented reality device.

Enhanced Real-Time Machine Inspection [12] is an inspection system for an industrial worker that improves the worker's productivity, safety and effectiveness exploiting Hololens and AR. Similarly, [13] analysed the users' reaction and feedback on various AR interfaces in order to come up with an unique design that is natural and fits to a diverse category of users.

V. Conclusions

This study presented a hologram-based framework for the manipulation and control of IoT devices in industrial settings. We built an end-user centered architecture in which multiple IoT device were managed by a single edge board and controlled via holograms. We evaluated our system via a user study comparing the hologram to the conventional SCADA web application. The results revealed that users favor interaction via hologram. Our system benchmarks also revealed the limitations of existing HMD devices that deserve future investigation from the community.

References

[1] T. M. Fernández-Caramés and P. Fraga-Lamas, "A review on human-centered iot-connected smart labels for the industry 4.0," *IEEE Access*, 2018.

[2] K. B. Swain, G. Santamanyu, and A. R. Senapati, "Smart industry pollution monitoring and controlling using labview based iot," in *Sensing, Signal Processing and Security (ICSSS), 2017 Third International Conference on*. IEEE, 2017.

[3] Virtual Sensor opens a World of Efficiency for Large Motors. Siemens. [Accessed: 15.10.2018]. [Online]. Available: https://sie.ag/2zbcn47

[4] A. Tang, C. Owen, F. Biocca, and W. Mou, "Comparative effectiveness of augmented reality in object assembly," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2003.

[5] R. W. Sperry, "Hemisphere deconnection and unity in conscious awareness." *American Psychologist*, vol. 23, no. 10, p. 723, 1968.

[6] J. J. Paton, M. A. Belova, S. E. Morrison, and C. D. Salzman, "The primate amygdala represents the positive and negative value of visual stimuli during learning," *Nature*, vol. 439, no. 7078, p. 865, 2006.

[7] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui, "Mobile augmented reality survey: From where we are to where we go," *IEEE Access*, vol. 5, pp. 6917–6950, 2017.

[8] J. Wither, S. DiVerdi, and T. Höllerer, "Annotation in outdoor augmented reality," *Computers & Graphics*, vol. 33, no. 6, pp. 679–689, 2009.

[9] M. Wich and T. Kramer, "Enhanced human-computer interaction for business applications on mobile devices: a design-oriented development of a usability evaluation questionnaire," in *System Sciences (HICSS), 2015 48th Hawaii International Conference on*. IEEE, 2015.

[10] K. Michalakis, J. Aliprantis, and G. Caridakis, "Visualizing the internet of things: Naturalizing human-computer interaction by incorporating ar features," *IEEE Consumer Electronics Magazine*, 2018.

[11] M. Berning, T. Riedel, D. Karl, F. Schandinat, M. Beigl, and N. Fantana, "Augmented service in the factory of the future," in *Networked Sensing Systems (INSS), 2012 Ninth International Conference on*. IEEE, 2012, pp. 1–2.

[12] M. Jayaweera *et al.*, "Enhanced real-time machine inspection with mobile augmented reality for maintenance and repair: Demo abstract," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*. ACM, 2017.

[13] G. Alce *et al.*, "Ar as a user interface for the internet of things-comparing three interaction models," in *Mixed and Augmented Reality (ISMAR-Adjunct), 2017 IEEE International Symposium on*. IEEE, 2017.

Publication VI

Cozzolino, V., Ding, A. Y., & Ott, J. (2019). Edge chaining framework for black ice road fingerprinting. In A. Y. Ding, & R. Mortier (Eds.), EdgeSys 2019 - Proceedings of the 2nd ACM International Workshop on Edge Systems, Analytics and Networking, Part of EuroSys 2019: EdgeSys 2019 (pp. 42-47). Association for Computing Machinery (ACM). https://doi.org/10.1145/3301418.3313944

Publication Summary

The presence of a smart roadside infrastructure in the cities of tomorrow offers opportunities for building new applications working with and providing fine-grained, localized information. Consequently, more detailed and precise maps of metropolitan areas can be generated to support applications in, for instance, the health and safety domains. City-wide pollution fingerprinting can enable pedestrians and cyclists to select less polluted routes, while infrastructure-supported black ice detection can allow drivers to predict the presence of patches of black ice outside their field of view.

We focused on the detection of road condition hazards which is today a challenging task given practical restrictions such as limited data availability and lack of infrastructure support. We presented an edge-cloud chaining solution that bridges the cloud and road infrastructures to enhance road safety. We exploited the roadside infrastructure (e.g., smart lampposts) to form a processing chain at the edge nodes and transmit the essential context to approaching vehicles providing what we refer as road fingerprinting. We approached the problem from two angles: (i) semantically defining how an execution pipeline spanning edge and cloud is composed, and (ii) we designed, implemented and evaluated a working prototype based on our assumptions. We presented our experimental insights and outlined open challenges for next steps.

Author's Contribution

I came up with the idea for the paper as a foundation for the applicability of our unikernel framework in the domain of black-ice detection. I have designed, implemented, and evaluated the entire system.

Edge Chaining Framework for Black Ice Road Fingerprinting

Vittorio Cozzolino
Technical University of Munich
Munich, Germany
cozzolin@in.tum.de

Aaron Yi Ding
Delft University of Technology
Delft, Netherlands
aaron.ding@tudelft.nl

Jörg Ott
Technical University of Munich
Munich, Germany
ott@tum.de

Abstract

Detecting and reacting efficiently to road condition hazards are challenging given practical restrictions such as limited data availability and lack of infrastructure support. In this paper, we present an edge-cloud chaining solution that bridges the cloud and road infrastructures to enhance road safety. We exploit the roadside infrastructure (e.g., smart lampposts) to form a processing chain at the edge nodes and transmit the essential context to approaching vehicles providing what we refer as road *fingerprinting*. We approach the problem from two angles: first we focus on semantically defining how an execution pipeline spanning edge and cloud is composed, then we design, implement and evaluate a working prototype based on our assumptions. In addition, we present experimental insights and outline open challenges for next steps.

CCS Concepts • **Computer systems organization → Distributed architectures**; *Embedded and cyber-physical systems.*

Keywords Edge Computing, Distributed Systems, IoT

ACM Reference Format:
Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2019. Edge Chaining Framework for Black Ice Road Fingerprinting. In *2nd International Workshop on Edge Systems, Analytics and Networking (EdgeSys '19), March 25, 2019, Dresden, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3301418.3313944

1 Introduction

The presence of a smart roadside infrastructure in the cities of tomorrow (e.g. smart lampposts [3]) offers opportunities for building new applications working with and providing fine-grained, localised information. Consequently, more detailed and precise maps of metropolitan areas can be generated to support applications in, for instance, the health and safety domains. City-wide pollution fingerprinting can enable pedestrians and cyclists to select less polluted routes, while infrastructure-supported black ice detection can allow drivers to predict the presence of patches of black ice outside their field of view.

Numerous applications fully exploit crowdsourcing to generate *augmented* maps which, however, have limited dimensionality in terms of collected data. In fact, they rely on users' hand-held devices, which may not be equipped pollution or particulate sensors or infra-red (IR) thermal cameras, which are fundamental for the aforementioned functions.

EdgeSys '19, March 25, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6275-7/19/03…$15.00
https://doi.org/10.1145/3301418.3313944

As smart cars become widespread, the availability of sensors will expand the information domain of crowdsourcing applications; however, there are multiple limitations. First, data availability: data collected by car manufacturers is often not publicly available, thus, creating a disparity in the quality of service offered among providers. Second, bad weather conditions, limited range of car sensors and lack of enough reliable data to properly map the road condition can reduce the effectiveness of crowdsourcing solutions and eventually provide false predictions putting at serious risk the drivers. This problem is exacerbated in areas with poor network connectivity where vehicles are hardly reachable from the cloud and left in the dark about the presence of road hazards. Section §2 will provide a more detailed discussion about such issues.

The problem we wish to solve concerns *how to provide reliable information regarding road hazards to vehicles in challenging conditions with the support of a smart roadside infrastructure.* Our primary use case is black ice detection, which we tackle with an *edge-cloud pipelining* concept to create on-demand execution pipelines spanning edge and cloud nodes. Involved edge nodes (ENs) form execution chains and follow a specific protocol in order to collaboratively contribute to the task completion. Assessed road conditions are then broadcast to approaching vehicles. In this paper, we build upon our previous groundwork [7], develop a new edge chaining framework, and contextualize it for the above-mentioned use-case.

The remainder of this paper is structured as follows: framing the problem (§2), background (§3), edge-cloud pipeline (§4), system design (§5), implementation (§6), evaluation (§7), conclusion and future work (§8).

2 Framing the Problem

With the growth in deployed smart devices and the presence of physical infrastructure in proximity to end-users, there arises the challenge of constructing a platform that can provide **accurate**, **reliable** road conditions information at **scale**.

Detecting road conditions and potential hazards is a problem that has been explored and approached in the literature using different approaches. Both crowdsourcing solutions, where vehicles exchange collected information to identify bumps [5], and infrastructure based solutions as in [13], where IR cameras mounted on lampposts are used to detect ice formations on the road, have been explored. Through different approaches and tools, various studies examining the effectiveness of detecting road conditions have been conducted [11, 12].

Eriksson et al. [8] proposed pothole patrol (P2), a mobile sensing application used for detecting and reporting road surface conditions. In a similar system used in traffic sensing and communication, Mohan et al. [16] proposed the use of mobile devices connected to exterior sensors. Mednis et al. [15] improved and extended the P2 system using a customised embedded gadget and with the aid of a smartphone hardware platform for sensing road surface conditions [21]. Edge-computing-based approaches have been also explored,

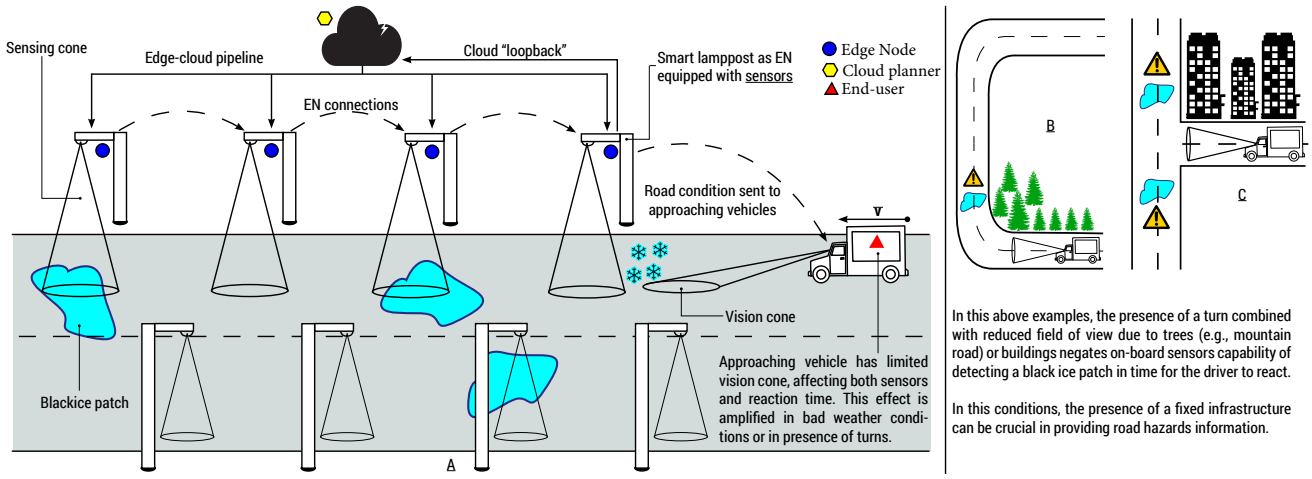


Figure 1. Black ice road fingerprinting (A); critical scenarios (B, C)

as in [5]. However, their focus is on security implications, while the focus of this study is to formalize the system requirements and build a working distributed edge computing platform.

Existing solutions focus on using either crowdsourcing or edge networks for transferring information. However, the quality of crowdsourced spatial data is often unreliable [6]. Consequently, the density of effective data points for estimating road conditions can be insufficient in low-traffic areas. Moreover, solutions based on on-board car sensors are also unsatisfactory in circumstances where road characteristics (e.g. buildings, trees, turns, crossroads) and adverse weather conditions effectively inhibit the ability to detect hazards at a distance. To overcome this challenge, we take advantage of road infrastructure by extending the sensory capacity of cars beyond what can be captured by on-board sensors, and use edge computing as a core technology.

In this paper, first, we tackle the problem of semantically representing a distributed task that spans multiple ENs. Then, we introduce the definition of an edge-cloud pipeline and describe the process of splitting it into local sequences. Finally, we identify the required software components and emerging technologies that can fulfil the desired functions.

3 Background

Here, we introduce several concepts and definitions that are used throughout the remainder of the paper.

Edge node (EN). The definition og this term is quite broad. We agree with the definition provided in [20], in which edge computing generally occurs in proximity to datasources. Hence, an EN is a device very close to the end user, such as a base station, mobile phone or private PC. Other classifications of ENs extend the definition to RAN microservers [14]. In this work, we focus on ENs in the range of micro-servers such as Intel NUC and Dell Optiplex which were used in our experiments.

Edge network. This is a network of ENs interconnected via a wired or wireless connexion. The ENs are in physical proximity to one another, such as lampposts on the side of a road. Detailed characteristics of an edge network are not described here, as they are discussed in other papers.

Task and edge function (EF). In this paper, we use the term *tas* to refer to an operation to be carried out by the network. At a high level, a task can be expressed as follows: *'find black ice patches at road intersection 1A and 3B'*. A task can be reduced to an ensemble of EFs: self-contained, *atomic* applications in which a small fraction of the task logic is embedded, but can be executed in a standalone fashion. A task contains at least one edge function. Distributed complex event processing [18, 19] is an example of a task composed of multiple sub-functions. To improve performance and scalability, these sub-functions are moved closer to the data sources.

Edge-cloud pipeline. This pipelines is a tasks issued by a cloud provider to edge nodes, and contains information regarding involved nodes, chaining order and data sources. Nodes involved in the pipeline form execution chains and collaborate to solve the task; they are selected based on multiple parameters and execute only a subsection of the entire pipeline. Pipelines can be ephemeral or recurrent, based on the task requirements. More details are provided in §4.

4 Edge-Cloud Pipeline

In this section, we describe in detail our representation of the edge-cloud pipeline and its application to our prime use case: blackice road fingerprinting. Three main elements are involved: the cloud planner, the edge infrastructure and the vehicles. Due to page limit, the vehicles are not discussed in details as their role is passive in relation to our system — they receive the information from the infrastructure via, for instance, long-range communication radios such as LoRaWAN [2] or Vehicle Fog Computing [10].

4.1 Cloud planner

Cloud services define the pipeline structure and monitor its execution. We assume knowledge is available regarding th reachability of edge devices, their available data and their current load (in terms of active EFs and pipelines). Based on this assumption, it is possible to plan a pipeline execution tree based on a set of parameters among which *data locality* has a prime role. Once offloaded, the pipeline can be configured to run independently from the cloud based on specific policies.

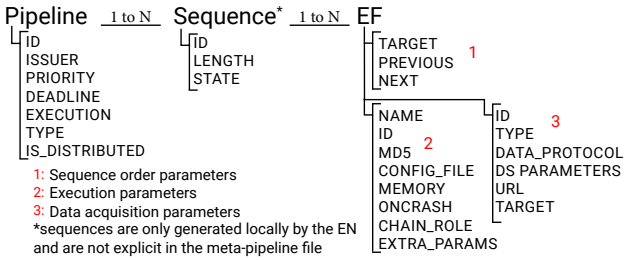


Figure 2. Meta-pipeline

Once the structure of the pipeline is defined, the cloud prepares a meta-pipeline containing various information about the pipeline itself, as illustrated in Figure 2. In our use case, the cloud provider may be a car manufacturer that wishes to offer augmented maps to its fleet [1] and, thus, exploits edge infrastructure to collect and analyse road condition data.

4.2 Edge infrastructure

In relation to our use case, black ice detection requires each node to locally process thermal images acquired from IR cameras or similar sensors, identify patches of black ice and send the results to the following node. As Figure 1 shows, we assume that ENs are deployed inside smart lampposts that are equipped with an array of sensors able to detect road conditions.

The edge infrastructure is composed of manifold ENs each addressable by an unique identifier such as their location (e.g. GPS coordinates). When a meta-pipeline is offloaded, the involved ENs parse it and identify which sections has to execute and in which order relative to the other nodes. Each pipeline is split into sub-pipelines, which in turn are transformed into sections that can contain multiple stages, which are eventually, but not immediately, executable. Pipelines can be sorted by multiple parameters: priority, expected load and deadline which can alter the execution order.

Once the sub-pipeline execution order is assessed, each EN has to verify the reachability of the neighbour ENs in the pipeline and prepare to exchange and process data with them (here we assume that there are no issues in terms of network reachability).

4.3 Pipelines: flexible chaining at the edge

One concept that requires further explanation is the relationship between pipeline and sequence. As mentioned above, ENs scan the meta-pipeline and select the list of EFs to be executed locally. These EFs can have different roles, such as *head*, *transit* and *tail*. This classification is necessary because it determines the order in which the EFs should be chained and executed. The ordering is based on the identifiers associated with each EF. Identifiers (IDs) are assigned to pipelines, sequences, EFs and data acquisition rules, as demonstrated in Figure 2. *Head* and *tail* roles always represent the end of a sequence, while transits are EFs that serve as links between the head and tail. In other words, a sequence always begins with a head and ends with a tail; they are not necessarily deployed on the same EN, and there can be an arbitrary number of transit instances between the two. *Sequence order parameters* are used to correctly unfold the execution. Figure 4 illustrates different pipeline and sequence structures.

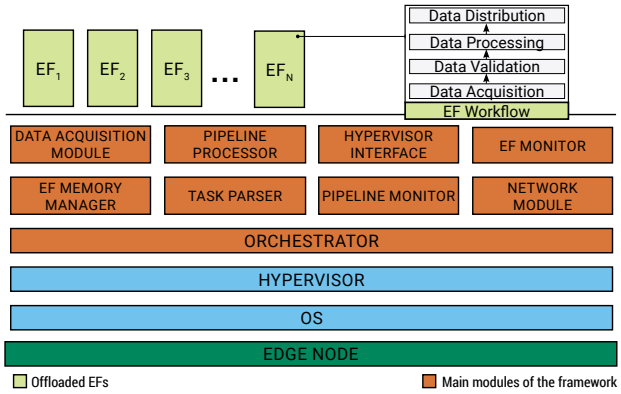


Figure 3. System modules overview

A key property of the pipeline is **chaining**. ENs create temporary, dynamic execution chains based on the pipeline topology in order to form a collaborative network. As data flows from one link of the chain to the other, computation unfolds and the pipeline progresses towards the *tail* EN. In addition, pipelines are not necessarily linear, they can *branch* and *join* creating execution trees. For instance, branching may be required in situation where groups of ENs provide different information to be subsequently merged at the end of a pipeline.

Each EF has data acquisition rules that can be split into input and output. The former determines where the data should be retrieved, while the latter specify whether the result of the computation should be sent to a remote node to continue along the pipeline or should be stored locally for future use. Additional details concerning this process are provided in section §5.

5 System Design

In this section, we provide an overview of the system (as presented in Figure 3) in relation to the discussed use case followed by a description of the core components necessary for pipelining. To conclude, we briefly describe the system workflow.

5.1 Overview

In the scenario in which roadside infrastructure is used exclusively as an adjunct of the cloud, a slow connexion or the lack thereof can invalidate the purpose of the entire system as follows: information about the road conditions is slowly retrieved or not retrieved at all, and vehicles are left without information about potential hazards, which can potentially cost lives. Hence, edge computing has the critical role of being a **reliable, resilient and autonomous infrastructure** that delivers services even when the cloud is unreachable.

As vehicle density on side-roads might not be sufficient to map reliably the road conditions, crowdsourcing in unable tackle this problem. Available spatial data is very sparse and limited, leading to incomplete or misleading information distributed to vehicles driving in low-traffic areas. In terms of on-board car sensors (when available), there are two key situations in which their effectiveness is hampered: harsh weather conditions where visibility is heavily reduced (e.g. blizzard, hailstorm, fog), or the scenarios illustrated in Figure 1 (B,C). In both cases, the vehicle and driver fields of view

are reduced, limiting the sensor detection capabilities and reaction time, respectively.

Our framework is designed to solve the aforementioned challenges, as it provides a framework where multiple cloud services can share existing infrastructure, which in turn facilitates scheduling and handling multiple offloaded tasks. It offers computation at the ENs, enabling both independence from the cloud in case connectivity loss and efficient processing and aggregation of information based on EFs chaining. In addition, local broadcasting of road hazards to approaching vehicles affects end-users, who can benefit from a standalone infrastructure without requiring a mobile connexion.

5.2 Core components

Our system pivots around the concept of edge offloading [17]: an emerging paradigm by which computation can be moved from the cloud towards edge nodes in order to provide multiple benefits. To differentiate from similar solutions, we design our system as a collaborative framework where multiple ENs can be chained to execute different tasks. In order to support and manage the offloaded EFs, we developed a set of modules residing and constantly running directly on each EN.

Orchestrator. It's the core and entry point of our system, and it functions as both a coordinator and interface with the outside world. A REST interface is employed to interact with it. When one or more EFs are offloaded as part of a pipeline, the orchestrator handles the calling of the required modules to philtre, order and execute the EFs. Both single execution and recurrent pipelines are supported. For resiliency purposes, checkpoints of the pipeline status plus EFs intermediate results are stored in a local database.

Edge function. The EFs are virtual instances moved from the cloud to edge devices as part of the pipeline. They are composed of four parts: data acquisition, validation, processing and distribution. During the first phase, an EF awaits the necessary data from the orchestrator (intra-node communication). Next, it proceeds to the validation phase, in which received data is checked for errors in case of faulty transmission, eventually requesting a re-transmission. The data processing phase contains the developer code and is the core of the EF. By customising this part of the EF, it is possible to execute any computation inside the EF, granted that eventual external dependencies and libraries have been handled. Finally, the distribution phase contains rules that specify whether outputted data should be dumped to the host or sent to the next EF in the local sequence.

Intra-node and inter-node communication. Based on the pipeline structure, data can be exchanged in two ways. Intra-node communication occurs when the transfer involves two co-located EFs or an EF and the orchestrator. In contrast, inter-node communication occurs when the transfer takes place between two EFs that are not co-located. For the former, a set of parameters is provided for identifying the source and destination addresses of shared memory pages blocks used to transfer the data. When two co-located EFs are involved, the transfer is performed automatically and requires no involvement from the host modules. In other cases, the host *memory manager* module has the task of allocating, deallocating and cleaning up the memory pages that are continuously used to communicate with each EF. EFs are unable to fetch data from a local or remote source; they exist in a completely sealed environment. Hence, the host framework must also handle the collection of the required data from the data source specified in the respective meta-pipeline section.

Network module. This module enables the communication between non-co-located ENs. It has two groups of queues containing data structures called *bundles*, which contain a set of parameters used to unequivocally identify a pair of producer and consumer ENs. A combination of IDs extracted from the meta-pipeline is used for this purpose. The bundles in the inbound queue are stored until consumed by one or more local EFs which remain in a waiting state until the specific data is available. The outbound queue contains the bundles that are ready to be forwarded to the next EN in the pipeline. The outbound queue is also necessary in case of failures in some stages of the pipeline; data bundles are in fact stored until the malfunctioning nodes are prepared to continue. For additional resiliency, the bundles are also stored inside a database to allow hot restarts of the system in case of local failure.

5.3 Workflow

The cloud service sends the generated pipeline to each selected EN. The orchestrator extrapolates relevant information from the meta-pipeline file with the *task parser* module. Next, the *pipeline processor* generates the local sequences and boot-up the required EFs while the *pipeline and the EF monitor* supervise the status of the respective components. Depending on the data provenance, either the *data acquisition module* or the *network module* prepare a data bundle subsequently passed to the *EF memory manager*. As the data starts flowing, the computation takes place with EFs being allocated and deallocated following the prescribed order. Once the local sequence is complete, collected data is either sent to the approaching vehicles, in case we reached the end of the pipeline, or sent to the next EN.

6 Implementation

Our platform was developed as a unikernel-based system running on top of the Xen hypervisor (stock version). We exploited virtualisation for multiple reasons to hide hardware discrepancies between off-the-shelf devices, achieve fine-grained control over running VMs, obtain stronger isolation and maintain compatibility with existing cloud computing platforms. We considered Xen [4] to be the most suitable hypervisor for our implementation as it directly supports MirageOS [9] — the unikernel of choice for our implementation. We opted for this specific technology, since one of our goals was to offload compact, ready to launch, small virtual instances, embedding only the required code; unikernels are the perfect fit for this scenario.

Three languages were used to implement the entire system: C for the EF memory module (kernel module), Python for the majority of the core modules and OCaml for the EF code (MirageOS). Existing MiniOS and MirageOS libraries were modified and extended for compatibility with our system.

7 Evaluation

We profiled our system under different loads, devices and pipeline topologies, as shown in Figure 4. The basic case A represent black ice detection done by a pipeline involving only two lampposts, the results reflect the processing load of the edge computing nodes. Other cases grow in complexity and allow to profile our system under different configurations. Each pipeline begins and ends with a red and green block, respectively. Yellow blocks represent transit

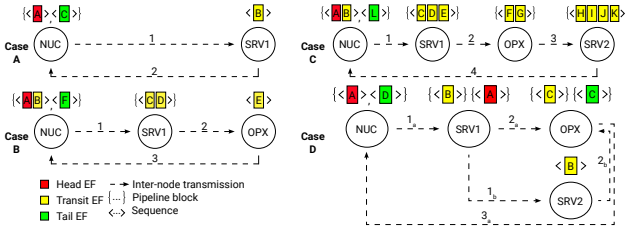


Figure 4. Pipeline topologies

EFs, which can be found in the middle of the chain. A pipeline block is delimited by curly braces and contains sequence blocks delimited by angular braces. A block is an EF processing images or other sensor data locally collected by the lamppost. Case D has two pipelines delimited by curly brace blocks. Each node executes only the blocks assigned to it, which are expected to be executed in a specific order (represented here with a character in each block).

Different devices were used to understand the performance gap between the edge and cloud. In our tests, the edge devices were comparable to micro-servers rather than base stations. The nodes used in our tests were an Intel NUC (NUC), Dell Optiplex (OPX) and two high-end Dell PowerEdge 730 servers (SRV1, SRV2), all connected to the same LAN network.

Table 1. Boot-up time for the EF unikernel on each device

NUC	OPX	SRV1	SRV2
46± 15 ms	*30± 13.2 ms*	*25± 7.4 ms*	*29± 11.5 ms*

We focussed on measuring the following parameters: intra-node data transfer overhead, pure computation time and pipeline completion time, as shown in Figure 5, 6 and 7 respectively. To this end, we baked a MirageOS unikernel supporting basic image processing operations such as colour normalization and continuously passed to it the same image with a size of approximately 250KB. Additionally, in our pipeline the nodes exchanged a complete post-processed image instead of a single value, as we expect in the case of black ice fingerprinting where a true/false is sufficient to at least communicate the presence of hazards. The average inter-node network transmission time is also specified for completeness: **20±0.039 ms**.

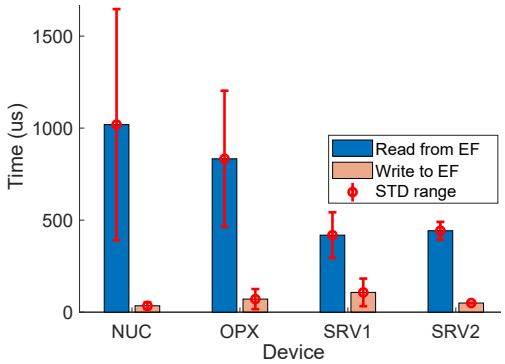


Figure 5. Intra-node transfer time (3000 data points evenly split between read/write per node)

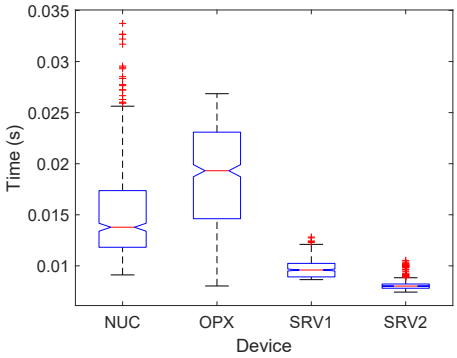


Figure 6. EF processing time (500 data points per node)

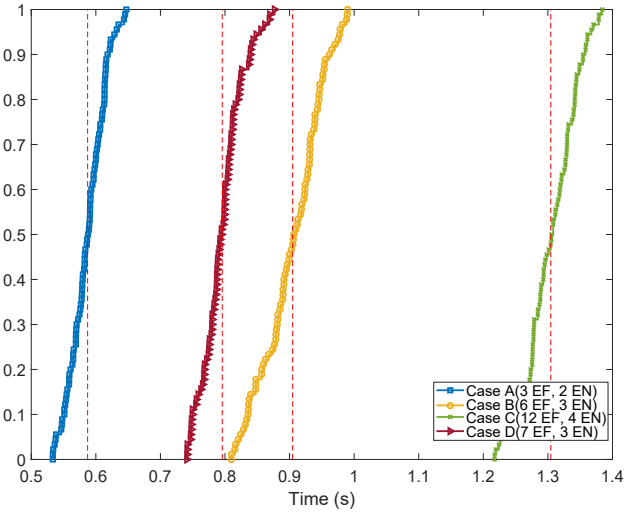


Figure 7. Pipeline execution time ECDF (100 iterations per topology)

In Figure 6, a minimal difference in performance can be seen between cloud (server) and edge devices. Thus, we can assert that there is an effective profit margin in offloading computation as the potential loss in computational speed is countered by a reduced upload time. In the case of black ice detection, it would be necessary to upload images from each lamppost to the cloud, process them and then send results to cars on a specific road. This would eventually increase the round-trip time and become unfeasible under in poor network connectivity situations.

Figure 5 illustrates the overhead created by transferring data between the host machine and guest unikernels before the processing phase. Both read and write operations are very fast, on the order of hundreds of microseconds. Read operations, in which memory pages populated with processed data are mapped back to the host, are much slower. This results from different code used for the two operations; writing is more efficient as we can allocate blocks of free memory pages directly, whereas the read operation requires proceeding page by page. However, even in the average worst case, reading data is in the range of **1ms**. Thus, the memory module design has very low overhead on the overall system performance.

Figure 7 displays the ECDF of the edge-cloud pipeline execution time for the cases presented in Figure 4. Cases A to C represent single-pipeline, non-branching scenarios with an increasing number of ENs and EFs. Case D represents the performance for a branched multi-pipeline scenario. The pipeline execution time includes the computational and memory overhead time in addition to the network inter-node transfer time and the unikernel boot-up time, as depicted in Table 1.

Our first insight into the results is the pipeline execution time grows linearly with the number of EFs. Even though they are not directly comparable, the first three topologies differ only in number of EFs and nodes, while the pipeline workflow remains roughly unchanged. We note that hardware differences do not play a major role, as performances is comparable. In each case, we double the number of EFs and add only one EN. By adding more nodes we amortise the overall completion time, thus, reducing the slope of the linear growth. This is critical in long pipelines, where finding the proper ratio of EF to EN has a large effect. Using our framework, to properly scale-up in terms of edge functions and nodes, different topologies can be tested to find the one with superior performance. In our scenario, EFs are expected to be quasi-uniformly distributed across the roadside equipment involved in the computation.

Case D is the sole case involving multiple pipelines. Rather than evaluation performance, the test serves as an example of the potential of our framework to run branching pipelines. The examples are situations where the set of information collectible by different EN is not uniform (e.g. smart lampposts equipped with different sensors). Hence, it is necessary to merge data collected from different branches of the pipeline to terminate the computation. In comparison to case B, whose computational load is the closest in terms number of EFs, we notice that there is a cost to running multiple parallel pipelines; with one additional EN, the results are only approximately 10% better than for the single pipeline case.

8 Conclusion and Future Work

In this paper, we proposed a distributed edge computing framework to improve current solutions to road hazard detection. Our attention was focused on black ice detection under adverse road conditions. The logic, design and implementation of our system were described in relation to with the analysed use case scenario. We discussed the advantages of our approach in comparison to solutions based on crowdsourcing, cloud computing and on-board car sensors. Our current evaluation has not yet demonstrated the full potential of our system, as the thermal image processing necessary for detecting patches of black ice was not embedded in the tested EFs. Hence, in future work we intend to address this issue by integrating a state-of-the-art machine-learning black ice detection model into our system.

To strengthen and expand the capability of our framework, we plan on offloading non-virtualised EF. Unikernels do not actually require a virtual hardware abstraction; they can achieve similar levels of isolation when running as processes by taking advantage of existing kernel system call whitelisting mechanisms as demonstrated in [22]. This has the potential to make our system compatible with a larger range of devices and enable a much simpler integration of existing tools into our platform. An alternative option is to modify our system to be compatible with a Tier-2 hypervisor such as KVM. In fact, this would allow us to compare our system with other similar frameworks for serverless computing, such as Amazon Firecracker.

References

[1] 2017. BMW Here HD Maps. https://www.forbes.com/sites/samabuelsamid/2017/02/21/bmw-here-and-mobileye-team-up-to-crowd-source-hd-maps-for-self-driving. [Online; accessed 08-January-2019].

[2] 2018. LoRaWAN. https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf. [Online; accessed 08-January-2019].

[3] 2018. Munich's Smart Lamp Posts Shine. https://www.smarter-together.eu/news/munichs-smart-lamp-posts-shine. [Online; accessed 07-January-2019].

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

[5] Sultan Basudan, Xiaodong Lin, and Karthik Sankaranarayanan. 2017. A privacy-preserving vehicular crowdsensing-based road surface condition monitoring system using fog computing. *IEEE Internet of Things Journal* 4, 3 (2017), 772–782.

[6] Alexis Comber, Linda See, Steffen Fritz, Marijn Van der Velde, Christoph Perger, and Giles Foody. 2013. Using control data to determine the reliability of volunteered geographic information about land cover. *International Journal of Applied Earth Observation and Geoinformation* 23 (2013), 37–48.

[7] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM, 36–41.

[8] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. 2008. The pothole patrol: using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*. ACM, 29–39.

[9] Madhavapeddy et al. 2013. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, Vol. 48. ACM.

[10] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. 2016. Vehicular fog computing: A viewpoint of vehicles as the infrastructures. *IEEE Transactions on Vehicular Technology* 65, 6 (2016), 3860–3873.

[11] Yoichiro Iwasaki, Masato Misumi, and Toshiyuki Nakamiya. 2013. Robust vehicle detection under various environmental conditions using an infrared thermal camera and its application to road traffic flow monitoring. *Sensors* 13, 6 (2013), 7756–7773.

[12] Maria Jokela, Matti Kutila, and Long Le. 2009. Road condition monitoring system based on a stereo camera. In *Intelligent Computer Communication and Processing, 2009. ICCP 2009. IEEE 5th International Conference on*. IEEE, 423–428.

[13] M Kutila, M Jokela, J Burgoa, A Barsi, T Lovas, and S Zangherati. 2008. Optical roadstate monitoring for infrastructure-side co-operative traffic safety systems. In *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE, 620–625.

[14] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. 2017. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2322–2358.

[15] Artis Mednis, Atis Elsts, and Leo Selavo. 2012. Embedded solution for road condition monitoring using vehicular sensor networks. In *the 6th IEEE International Conference on Application of Information and Communication Technologies (AICT'12)*. 1–5.

[16] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran Ramjee. 2008. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 323–336.

[17] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott. 2018. Consolidate IoT Edge Computing with Lightweight Virtualization. *IEEE Network* 32, 1 (Jan 2018), 102–111. https://doi.org/10.1109/MNET.2018.1700175

[18] Omran Saleh and Kai-Uwe Sattler. 2013. Distributed complex event processing in sensor networks. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, Vol. 2. IEEE, 23–26.

[19] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS '09)*. ACM, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/1619258.1619264

[20] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[21] Girts Strazdins, Artis Mednis, Georgijs Kanonirs, Reinholds Zviedris, and Leo Selavo. 2011. Towards vehicular sensor networks with android smartphones for road surface monitoring. In *2nd International Workshop on Networks of Cooperating Objects (CONET'11), Electronic Proceedings of CPS Week*, Vol. 11. 2015.

[22] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels As Processes. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 199–211. https://doi.org/10.1145/3267809.3267845

Publication VII

Cozzolino, Vittorio, Nikolai Schwellnus, Jörg Ott, and Aaron Yi Ding. "UIDS: Unikernel-based Intrusion Detection System for the Internet of Things." In DISS 2020-Workshop on Decentralized IoT Systems and Security. 2020. 23 February 2020, San Diego, CA, USA ISBN 1-891562-64-9, doi: 10.14722/diss.2020.23008.

Publication Summary

With the rising popularity of Internet of Things (IoT), an increasing number of devices are being connected to the Internet. Most of these devices are resource-constrained and security is often regarded as an afterthought. In fact, vulnerabilities have been found in all types of related devices, ranging from cars to light bulbs. Moreover, such devices are oftentimes connected to the network by people having little knowledge about security or privacy concerns. We found that IoT devices are too resource-constrained to employ traditional security tools (virus scanner, etc.), which renders both edge computing and enterprise networks far more exposed to attacks.

In this work, we aimed at laying the foundation of the concept of composable security through unikernels with the latter embedding self-contained security functionality quickly deployed on-demand. This work represents a first step in such direction with a signature-based, low-footprint Unikernel Intrusion Detection System (UIDS). Our prototype is based on the IncludeOS unikernel, ensuring low resource utilization, high modularity, and a minimalist code surface. We evaluated the performance of our solution on x86 and ARM devices and compare it against Snort, a widely known network intrusion detection system. The experimental results show that our prototype effectively detects all attack patterns while using up to 2-3x less CPU and 8x less RAM than our baseline. To summarize, in this work we made two contributions: (i) a signature-based IDS, capable of detecting common Denial of Service (DoS), and port scan attacks, which can be deployed on resource-constrained devices with minimal overhead and memory footprint, and (ii) a comprehensive evaluation of our solution with different hardware and datasets against a well-known IDS tool.

Author's Contribution

I came up with the idea for the paper as a foundation for a unikernel - based intrusion detection system (IDS). I contributed to the system design aspects and research foundation while Nikolai Schwellnus implemented and evaluated the system.

UIDS: Unikernel-based Intrusion Detection System for the Internet of Things

Vittorio Cozzolino, Nikolai Schwellnus and Jörg Ott
Technical University of Munich
cozzolin@in.tum.de, n.schwellnus@tum.de, ott@in.tum.de

Aaron Yi Ding
Delft University of Technology
aaron.ding@tudelft.nl

***Abstract*—The advent of the Internet of Things promises to interconnect all type of devices, including the most common electrical appliances such as ovens and light bulbs. One of the greatest risks of the uncontrolled proliferation of resource constrained devices are the security and privacy implications. Most manufacturers' top priority is getting their product into the market quickly, rather than taking the necessary steps to build security from the start, due to high competitiveness of the field. Moreover, standard security tools are tailored to server-class machines and not directly applicable in the IoT domain. To address these problems, we propose a lightweight, signature-based intrusion detection system for IoT to be able to run on resource-constrained devices. Our prototype is based on the IncludeOS unikernel, ensuring low resource utilization, high modularity, and a minimalist code surface. In particular, we evaluate the performance of our solution on x86 and ARM devices and compare it against Snort, a widely known network intrusion detection system. The experimental results show that our prototype effectively detects all attack patterns while using up to 2-3x less CPU and 8x less RAM than our baseline.**

I. Introduction

With the rising popularity of Internet of Things (IoT), an increasing number of devices are being connected to the Internet. Most of these devices are resource-constrained and security is often regarded as an afterthought. Mirai [3], Qbot [6], and Torii [20] are examples of large-scale network attacks enabled by the proliferation of vulnerable, badly configured smart devices. The distributed and hardly controlled nature of the IoT transformed it into what is today a powerful cyberattack platform. In fact, vulnerabilities have been found in all types of related devices, ranging from cars [24] to light bulbs [10]. Moreover, such devices are oftentimes connected to the network by people having little knowledge about security or privacy concerns. In 2010, a study [8] found that 13% ($\geq$580.000) of the discovered embedded devices still used factory default login credentials. In 2017, Positive Technologies found around 15% of devices with factory-default credentials, which proved to be an exacerbation of the problem [1].

IoT devices are too resource-constrained to employ traditional security tools (virus scanner, etc.), which renders both edge computing and enterprise networks far more exposed to attacks [11]. Additionally, botched or corrupted updates can leave the device in an unstable state, which may be hard to recover from due to the lack of user interfaces. What we need are security tools that are lightweight, modular, and easily deployable. Hence, we aim at laying the foundation of the concept of *composable security* through unikernels with the latter embedding self-contained security functionality quickly deployed on-demand. This work represents a first step in such direction with a signature-based, minimalistic ***U**nikernel **I**ntrusion **D**etection **S**ystem* (UIDS), which can to deliver the same detection capabilities of well-known IDSs (e.g., Snort) while using **2-3x** less CPU and **8x** less RAM. We make two contributions:

- A signature-based IDS, capable of detecting common Denial of Service (DoS), and port scan attacks, which can be deployed on resource-constrained devices with minimal overhead and memory footprint.
- A comprehensive evaluation of our solution with different hardware and datasets against a well-known IDS tool.

II. Related Work

Intrusion detection is a very mature field of research going back to Anderson's "Computer security threat monitoring and surveillance" [2]. The first prototype of a real-time IDS was developed between 1984 and 1988, called the intrusion detection expert system (IDES) [9]. Currently, the rising number of deployed embedded devices and sensors has motivated researchers to explore IDSs that can run on resource-constrained devices. As there are few specific solutions targeting IoT networks, part of the research is focused on adapting and profiling desktop-class tools, such as Snort[1], to be less resource-intensive. One interesting study is [15], where the authors investigated the performance of Snort and Bro[2] on wireless mesh networks (WMNs). It was found that these IDSs are unsuitable as a security solution for WMNs as they are too demanding. To address this problem, they posited a lightweight IDS for WMNs that decreases memory consumption and packet drop rates in such resource-constrained nodes. However, it could only detect a few types of attacks. Similarly, in [14], the authors also argued about the infeasibility of deploying Snort in WMNs and proposed a distributed solution called PRactical Intrusion DEtection in resource constrained wireless

[1]https://www.snort.org/

[2]Today known as Zeek — https://en.wikipedia.org/wiki/Zeek

Workshop on Decentralized IoT Systems and Security (DISS) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-64-9
https://dx.doi.org/10.14722/diss.2020.23008
www.ndss-symposium.org

mesh network (PRIDE). Kyaw et al. [17] compared Snort and Bro IDS running on a Raspberry Pi 2, and showed that a Raspberry Pi 2 has enough resources to run open-source IDSs such as Snort or Bro sufficiently fast to detect DoS attacks and port scans. In addition the authors concluded that Snort performed better than Bro on the Raspberry Pi. For the IoT edge network, researchers have also used Docker containers [12] to deploy cloud-assisted security functionalities, especially for D2D communication [13]. Finally, [21] focused on the present and in-deep analysis of the feasibility of deploying an IDS infrastructure based on Snort and Raspberry PIs. Based on their results, this is possible in small networks; however, and more experiments are needed to better grasp the true limits of the Raspberry Pi. Another example of IDSs for constrained devices is CEPIDS [5], which is a complex event processing (CEP)-based IDS for detecting DoS attacks and port scans. CEPIDS follows a similar architecture to Snort. It uses three components to collect, evaluate and potentially block malicious network traffic. The authors showed that their IDS performs better than Bro and is on-par with Snort. However, the methodology used to evaluate their solution is unclear, as they compare their results with those obtained from [17] by using a different dataset and device.

III. Background

In this section, we briefly introduce two main types of IDSs: signature-based and anomaly-based.

Signature-based IDSs use parameters of known attacks to detect them. Hence, one downside is that new attacks cannot be detected as long as their signatures are not yet known. As such, signature-based IDSs need to receive constant updates to be competitive. In addition, attacks that cannot be easily described with signatures are difficult to detect. On the other hand, a big advantage of signature-based IDSs is that known attacks can be detected fast, accurately, and with fewer *false positives*. This approach, however, depends on the accuracy and quality of the signatures. If the signatures are known, an attacker can craft traffic that is benign but triggers signature-based rules, classifying the traffic as malicious, and as such generates many false positives.

Anomaly-based IDSs require a *normal operation* model against for comparison to the current network traffic. The flowing traffic characteristics are then compared to this baseline, and if an anomaly is found, the IDS will generate an alert.

In the case of network-based IDSs, machine learning is oftentimes used to build a model trained on non-malicious traffic. Incoming packets not fitting the model are classified as abnormal and an alert is generated. An obvious problem with this approach is that no malicious traffic must be present during the model learning phase; otherwise, malicious traffic will be classified as normal and no alerts will be generated.

Anomaly-based approaches can detect attacks that are unknown at the time of deployment. However, they depend heavily on the accuracy of the baseline model and require a potentially long and tedious training process. Hence, one of the main challenges in highly heterogeneous network traffic is building such accurate models.

IV. UIDS

We design UIDS as a signature-based IDS capable of detecting common DoS attacks, such as TCP SYN flood, TCP ACK flood, and UDP flood. Moreover, our signature based approach can detect the most common port scans in three different variations: *one-to-one*, *distributed* and *decoy* scans. We base our prototype on the IncludeOS [4] unikernel which follows the *zero-overhead* principle and is written in C++. UIDS expands on the rudimentary connection tracking capabilities of IncludeOS to classify traffic as suspicious or benign and keeps additional state information regarding possible malicious packets.

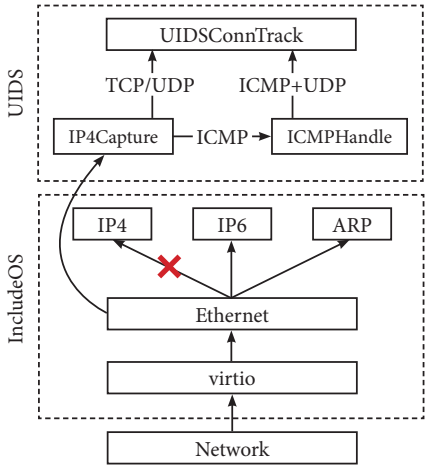


Fig. 1. UIDS packet reception path.

IncludeOS contains a few features that we found handy during the UIDS development. It offers state-keeping for network connections, UDP and ICMP, and a more sophisticated one for TCP. In addition, the IncludeOS modular network stack allows us to easily capture packets on the wire and redirect them to custom modules for additional processing. The network stack of IncludeOS comprises C++ classes for each module of the stack, such as IPv4, IPv6, and TCP which are connected together using delegates and can be rewired at runtime.

We extend IncludeOS as shown in Figure 1 to parse the captured packets and detect attacks. Packets received by the *virtio* device are passed up in the network's stack hierarchy of IncludeOS. After the Ethernet layer, we redirect packets to a custom capture module bypassing the standard one as shown in the figure with a red cross. Subsequently, we forward them to the core of our system: the *UIDSConnTrack* module. ICMP packets of type *destination unreachable* are parsed by *ICMPHandler* to extract the UDP packet that generated the ICMP error message. Afterwards, they are forwarded to the connection tracking module with the augmented data.

Port scans are classified as probes hitting different ports on the same host (vertical scans) or the same port on different hosts (horizontal scans). For this reason, we track suspicious packets on a per-host and per-port basis. UIDS classified packets as probes (suspicious) if at least one of the following criteria was met: *(i)* SYN packet to a closed, filtered or inactive port/host, *(ii)* ACK or FIN packet not belonging to an active connection, *(iii)* invalid packets (NULL/XMAS scan), *(iv)* partial three-way handshake, *(v)* UDP packets generating ICMP unreachable replies, and *(vi)* unanswered UDP packets.

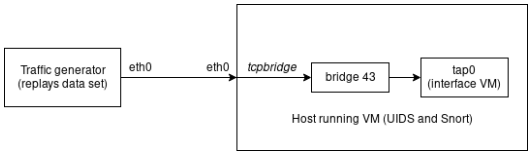


Fig. 2. Setup for traffic replay.

The *UIDSConnTrack* module stores information about packets that satisfied the above rules on a per-host, per-port basis using trackers. The latter are implemented as unordered maps, saving the address of the sending host, in addition to the scan type and time.
DoS detection is implemented similarly to port scan but with a different ruleset. In this case, trackers are simple packet counters, as common practice for such attacks, and store less information about the sender to save on resources.
The data structures tracking suspicious packets are analyzed periodically, and if an attack is recognized during a scan, the system generates an alert in the form of a JSON message. Subsequently, alerts are forwarded to an alert module, which either sends the data using UDP over a second network interface or flushes it to *stdout*.

V. Evaluation Setup

We benchmark UIDS attack detection capabilities against different datasets normally used to evaluate the effectiveness of IDSs. These are often developed to train and test anomaly-based IDSs using machine-learning but could also be used to evaluate signature-based IDS as well. In our evaluation, we use three publicly available datasets containing DoS attacks and port scans in a packet-based format. One of the most widely used dataset is the 1998 DARPA Intrusion Detection Evaluation Dataset[3]; however it is very old, and therefore did not contain traffic one would likely see today. In addition, several researchers have shown different flaws in this dataset [18], [19]. Hence, we instead used TRAbID [16] and CICIDS 2017 [22]. In addition to existing datasets, we use a small-scale testbed of our design to stress-test UIDS. Both port scans and DoS flooding attacks are used to stress-test our implementation and evaluate the accuracy of alerts raised by UIDS and Snort.

Traffic replay. UIDS and Snort run on KVM with QEMU using bridge networking to expose an interface to replay traffic to. IncludeOS comes with a deployment tool for KVM and QEMU, which allowed to configure this bridge networking. We replay the dataset network traffic through an Ethernet interface on the *traffic-generator* node and send it to the device hosting UIDS and Snort via the incoming network interface. On the receiving host, traffic is forwarded to the bridge interface using the tool *tcpbridge* included in the *tcpreplay* tool suit. Finally, the bridge interface (*bridge43*) is connected to the virtual machines (VMs). Figure 2 illustrates the complete network flow.
CICIDS2017 traffic is split into port scan and DoS to dramatically reduce the evaluation period from more than 9h to 1.5. However, such a procedure might introduce false-positives in the first few minutes of the new traffic since some connections might have been established directly before the split. Nevertheless, these false-positives were easily filtered out from the analysis (if they occurred).
The TRAbID dataset provides two traffic captures for port scan and DoS. We use the *probe_known_attacks* capture for the evaluation of the port scan detection accuracy. It contains seven port scans originating from different machines, but all directed to the same host. However, the amount of attack traffic is very small compared to the background traffic ($\leq$0.15%). Unfortunately, for this dataset, the DoS traffic capture is not currently publicly available, so we could not use it in our evaluation.

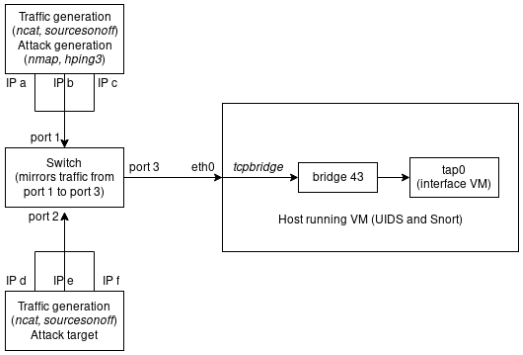


Fig. 3. Setup for live traffic.

Live traffic. Besides testing our solution against existing datasets, we also generate our own network traffic traces. For this purpose, we connect two hosts with a switch supporting port-mirroring. A host running UIDS and/or Snort is connected to the mirrored port, to receive all traffic generated between the hosts. Figure 3 illustrates the described setup.
To generate the traces, we uses the tool *sourcesonoff*[4], which outputs realistic Internet-like traffic using statistical models, detailed in [23]. Moreover, we develop a simple script using *netcat* to simulate an arbitrary number of concurrent TCP connections. Finally, the attack traffic is injected in the testbed using *nmap* for the port scan and *hping* for DoS traffic.

VI. Results

In this section, we present the results obtained by testing UIDS against the datasets described before while using Snort as the baseline. We focus primarily on CPU and RAM utilization to evaluate the compatibility of UIDS with resource-constrained devices. In terms of the memory footprint, UIDS weights only $\approx$2.3MB and boots in $\approx$200 ms on a non-optimized version of KVM (we did not use Solo5[5]). We run our tests on two different hardware platforms: a laptop equipped with an Intel i7-4710@2.5GHz (LAP) and a Raspberry Pi 3B+ with an A53 ARMv8@1.4GHz (RPI). On the former, both IDSs run virtualized on top of KVM. On the latter, due to the lack of support for ARM, IncludeOS can not be directly virtualized using KVM. Instead, we emulate the x86

[3]https://www.ll.mit.edu/r-d/datasets/1998-darpa-intrusion-detection-evaluation-dataset

[4]http://www.recherche.enac.fr/~avaret/sourcesonoff

[5]https://github.com/Solo5/solo5

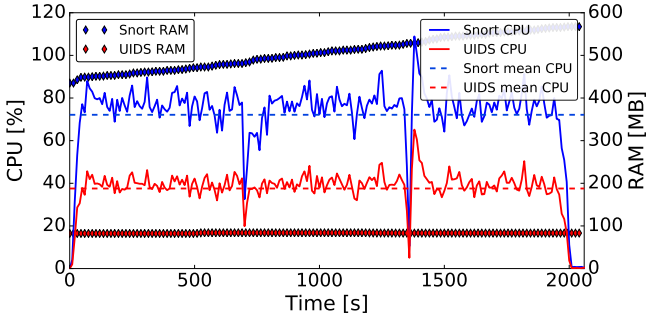


Fig. 4. UIDS vs. Snort — Port scan (TRAbID, LAP).

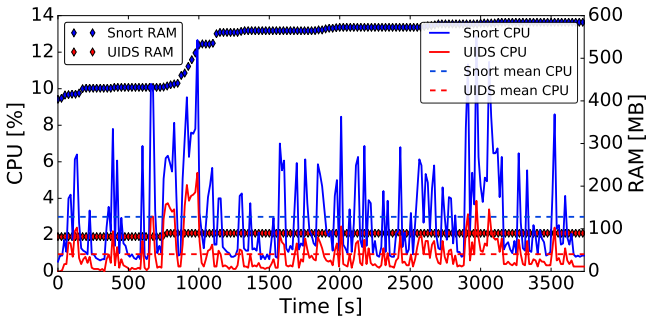


Fig. 5. UIDS vs. Snort — Port scans (CICIDS2017, LAP).

architecture on top of ARM using QEMU. Therefore, UIDS suffers a considerable performance penalty due to the additional emulation overhead which affected the results as well. Conversely, Snort runs baremetal, which gave it a considerable advantage in terms of performance. Hence, on the RPI we test UIDS in what could be seen as the worst case scenario.

A. TRAbID

Traffic from the TRAbID dataset is replayed over ≈33 min with an average speed of 33.68Mbps. Figure 4 shows the resource consumption on our laptop, which are correlated to the number of packets sent. Two dips in the graph at ≈600 and ≈1350 s are caused by a sharp drop in the number of packets generated by the dataset, moving from an average of ≥14000 to ≤1500 packets per second (pps).
The port scan traffic is not visible in the graph as it only represented a very small part of the overall traffic. Both UIDS and Snort equivalently detect 85% of all port scans contained in the dataset covering UDP, SYN, NULL, TCP connect, FIN, and XMAS scans. The ACK scan is not detected because it is not supported in UIDS and no rules to detect it are added to Snort. On the laptop, memory consumption for UIDS is just under 100 MB while it hovers around 400-600 MB for Snort. Additionally, UIDS CPU utilization is ≈40% while it is up to ≈80% for Snort. On the Raspberry Pi, both UIDS and Snort use, on average, the same amount of CPU. Regarding the memory allocation, UIDS is less demanding than Snort, requiring ≈50MB of RAM. Figure 7 shows the results in detail. Our solution is definitely penalized with this setup, as it is running on a twice virtualized stack, which clearly affects the performance.

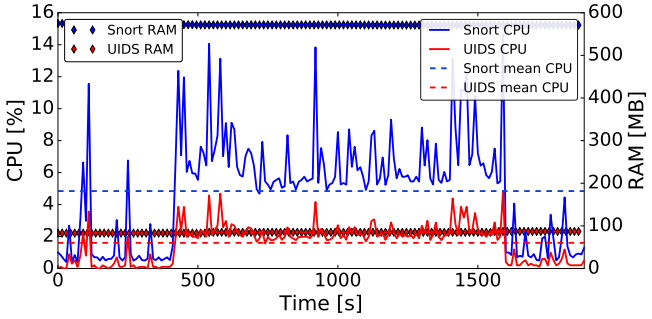


Fig. 6. UIDS vs. Snort — DoS (CICIDS2017, LAP).

TABLE I. CICIDS2017 — PORT SCANS AND RELATIVE DETECTION.

Scan	From (min)	Until (min)	UIDS	Snort
TCP SYN scan	11	13	yes	yes
TCP Connect scan	14	16	yes	yes
TCP FIN scan	17	19	no	no
TCP XMAS scan	20	22	no	no
TCP NULL scan	23	25	no	no
ICMP Ping scan	26	27	no	yes
TCP version scan	28	30	yes	yes
UDP scan	31	32	yes	yes
IP-protocol scan	33	35	no	no
TCP ACK scan	36	38	no	no
TCP window scan	39	41	yes	yes

B. CICIDS2017

The results for this dataset are divided for port scan and DoS attacks, and are described as follows.

Port scan. Port scans are executed during specific time windows, as specified in Table I. We notice that both UIDS and Snort detect most TCP/UDP-based scans contained in the dataset. As a side note, the CICIDS dataset supposedly contained FIN-, NULL-, and XMAS-scans, but could not find any evidence of such scans in the downloaded dataset. In fact, no packet without TCP flags set (NULL scan) or with URG, PSH and FIN flags set (XMAS scan) could be found using various tools. Therefore, neither UIDS nor Snort raise any alert regarding such attacks. The only difference in port scan detection between Snort and UIDS is the ICMP ping scan, which is not currently implemented, and therefore, is not detected by UIDS. The TCP version and window scans have similar characteristics as TCP SYN or connect scans, and are detected by both IDSs but classified as TCP SYN scans.
Snort and UIDS generated false positive alerts for FIN scans in the first 120 seconds of the dataset. As mentioned in Section V, this is due to the splitting of the dataset which led both systems to see finalization packets that belonged to connections lost during the splitting.
Figure 5 shows the resource utilization. Overall, the CPU usage is low for both IDSs because the packet rates in the CICIDS dataset are smaller than those in TRAbID, averaging around ≈330 *pps* and approximately 1 Mbps. Memory consumption is definitely higher for Snort, with 400-600MB, against UIDS, with less than 100MB (**4-6x** lower). A spike in memory usage can be observed after the first port scan is executed at the 11-12 min mark. Interestingly, this spike is modest for UIDS with a variation of ≤10MB but substantial for Snort with ≥130MB.

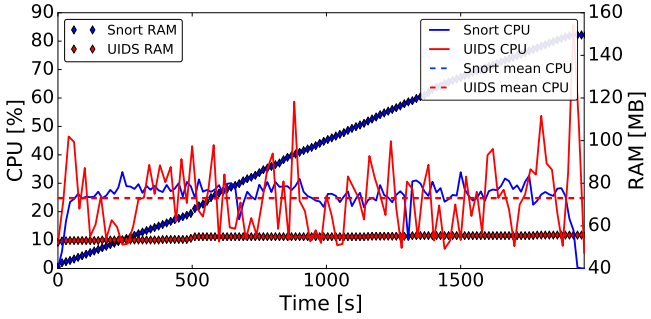


Fig. 7. UIDS vs. Snort — Port scan (TRAbID, RPI).

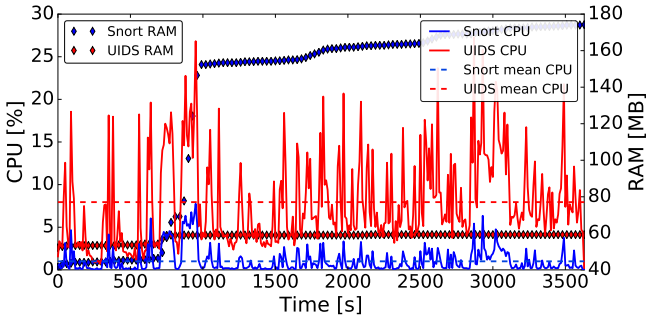


Fig. 8. UIDS vs. Snort — Port scan (CICIDS2017, RPI).

Flood-based DoS. Traffic from the CICIDS2017 dataset contains a DoS attack generated with the open-source tool Low Orbit Ion Cannon (LOIT)[6]. This attack is active from 15:56 until 16:16 and is contained in the second traffic slice generated by splitting the dataset. Both Snort and UIDS emit alerts correctly during the active phase of the attack. Figure 6 clearly shows the beginning and end of the DoS attack in relation to the number of CPU resources used. As foreseen, memory consumption remain stable and marginal during the attack since the very little state information needed storing to detect flood-based DoS attacks. Also, for this benchmark, UIDS proves to be extremely lightweight compared to Snort. In fact, it allocates ≈**4x** less memory than Snort during the attack peak and on average **3x** less CPU.

Figures 8 and 9 show CPU and RAM usage for the Raspberry Pi for CICIDS2017 port scan and DoS traffic, respectively. While UIDS CPU usage is **5-6x** higher compared with Snort, memory allocation is surprisingly reduced. Hence, considering that we are running in an emulated x86 environment, UIDS can handle moderately fast traffic (up to ≈34Mbps) for the CICIDS2017 dataset and reliably detect the same attacks as that in the case of running on more powerful hardware.

C. Results with custom testbed

We use our own testbed to evaluate the performance of UIDS under heavy loads, as described in section V. To simulate a fully utilized link, we open multiple TCP sessions with *netcat* and transmit random traffic as fast as possible through all concurrent connections. As the TCP protocol performs load

[6]https://sourceforge.net/projects/loic/

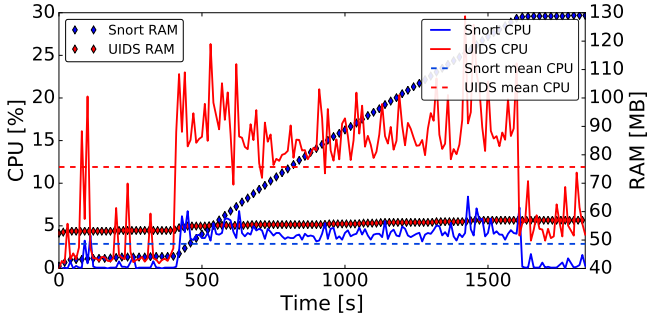


Fig. 9. UIDS vs. Snort — DoS (CICIDS2017, RPI).

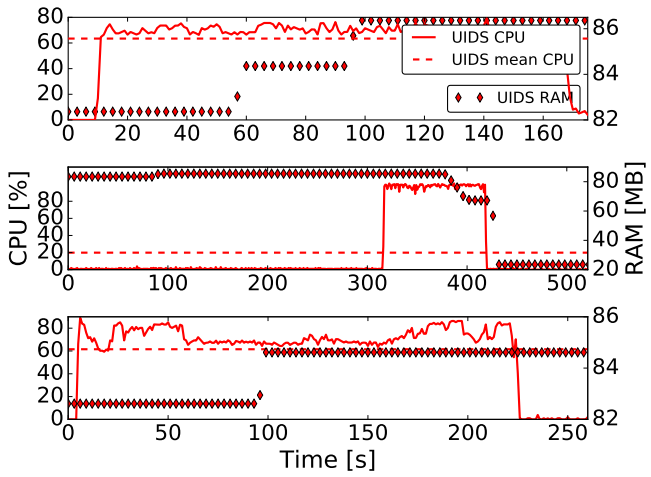


Fig. 10. UIDS stress-test (LAP) — Saturated 1Gbps link (top). Background traffic plus attacks (middle). 1Gbps, 5000 concurrent TCP connections (bottom).

balancing for us, we do not need any additional configuration steps. Figure 10(top) shows the resource consumption of UIDS dealing with a saturated 1Gbps link with an increasing number of concurrent connections, specifically, 1000 connections until 60 s, 5000 until 90 s and 10000 until 165 s. A step-up in memory consumption can be observed when the number of concurrent connections increased. This is expected, as UIDS needs to keep track of these extra connections.
Because the link capacity is fully utilized, some traffic is lost, and therefore, not available for our connection tracking algorithm. This is problematic because the port scan detector relied on accurate connection tracking. Moreover, we observe several false positives for TCP no-reply (i.e., lost answers to SYN packets) as well those for FIN scans.

In a second experiment, we evaluate whether UIDS could detect attacks in a realistic background traffic and if UIDS can cope with a large-scale DoS attack using 1Gbps traffic. Figure 10(mid) shows resource consumption during the background traffic, port scans, and a large DoS attack using the ICMP flood. The CPU usage is very small when handling traffic generated by the tool *sourcesonoff*, while memory usage is comparable to the results obtained for the other datasets.
Four different port scans are executed during the first 300 s of the second experiment: TCP SYN, TCP XMAS, TCP NULL and a UDP scan. All four scans are detected and no false alerts

are generated.
UIDS behaves as expected during the ICMP flood between 310 and 420 s, as shown in Figure 10(mid). However, the ICMP flood with ≈120,000 *pps* and close to 1Gbps rate is strong enough to effectively disable our traffic-generating host. Consequently, we see a reduction in the allocated memory as many connections timed out and no new ones are generated; thus, fewer connections need tracking.

Finally, we conduct a third experiment to evaluate the port scan detection capability under high load, as shown in Figure 10(bottom). We use the same four scans as in the previous experiment and a saturated 1Gbps link with 5000 concurrent TCP connections. In this case, all types of scans are detected including TCP SYN which got linked to false-positive alerts, and as such, are not accurate.

VII. Conclusions and Future Work

This paper presents UIDS: our first prototype of signature-based unikernel IDS for the IoT. UIDS is implemented from scratch in C++ and is based on IncludeOS. We evaluated our solution on both a mid-range laptop and a Raspberry PI and compared the results against Snort on two datasets. From our experiments, UIDS required **2-3x** fewer CPU resource and up to **8x** times less memory than Snort without penalizing any detection capability. We consider these results very promising as our main goal was to build a lightweight modular solution, with reduced hardware resource demand.
Despite this being our first prototype, UIDS showed great potential by delivering better resource efficiency, isolation, and a small memory footprint without sacrificing on the security aspects. Its modularity enabled easier code updates and opened the door to composable, on-demand security with unikernels. In our future work, we plan on exploring the tradeoffs of extending UIDS to anomaly-based detectors and simplifying its setup by using the IncludeOS configuration language NaCl and integrating UIDS with our edge-cloud deployment and chaining framework [7] to orchestrate a network of UIDS and fully exploit its modularity for service composition.

References

[1] "Positive technologies - learn and secure : Practical ways to misuse a router," http://blog.ptsecurity.com/2017/06/practical-ways-to-misuse-router.html, accessed: 2019-07-17.

[2] J. P. Anderson, "Computer security threat monitoring and surveillance," 1980.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[4] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, pp. 250–257.

[5] A. Cardoso, R. Fernandes Lopes, A. Teles, and F. Benedito Veras Magalhaes, "Poster abstract: Real-time DDoS detection based on complex event processing for IoT," 04 2018, pp. 273–274.

[6] G. Cluley, "Mutating Qbot worm infects over 54,000 PCs at organizations worldwide," *Tripwire, Tripwire*, 2016.

[7] V. Cozzolino, A. Y. Ding, and J. Ott, "Edge chaining framework for black ice road fingerprinting," in *Proceedings of the 2Nd International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '19. New York, NY, USA: ACM, 2019, pp. 42–47. [Online]. Available: http://doi.acm.org/10.1145/3301418.3313944

[8] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920276

[9] A. K. D and S. Venugopalan, "Intrusion detection systems: A review," *International Journal of Advanced Research in Computer Science*, vol. 8, 10 2017.

[10] N. Dhanjani, "Hacking lightbulbs: Security evaluation of the philips hue personal wireless lighting system," *Internet of Things Security Evaluation Series*, 2013.

[11] B. Duncan, A. Happe, and A. Bratterud, "Enterprise IoT security and scalability: how unikernels can improve the status Quo," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2016, pp. 292–297.

[12] I. Hafeez, A. Y. Ding, L. Suomalainen, A. Kirichenko, and S. Tarkoma, "Securebox: Toward safer and smarter iot networks," in *Proceedings of ACM Workshop on Cloud-Assisted Networking*, ser. CoNEXT CAN '16. ACM, 2016.

[13] I. Hafeez, A. Y. Ding, and S. Tarkoma, "Ioturva: Securing device-to-device (d2d) communication in iot networks," in *Proceedings of the 12th ACM MobiCom Workshop on Challenged Networks*, ser. MobiCom CHANTS '17. New York, NY, USA: ACM, 2017, pp. 1–6.

[14] A. Hassanzadeh, Z. Xu, R. Stoleru, G. Gu, and M. Polychronakis, "Pride: Practical intrusion detection in resource constrained wireless mesh networks," in *International Conference on Information and Communications Security*. Springer, 2013, pp. 213–228.

[15] F. Hugelshofer, P. Smith, D. Hutchison, and N. Race, "Openlids: a lightweight intrusion detection system for Wireless Mesh Networks," 01 2009, pp. 309–320.

[16] E. K. Viegas, A. Santin, and L. Soares de Oliveira, "Toward a reliable anomaly-based intrusion detection in real-world environments," *Computer Networks*, vol. 127, 08 2017.

[17] A. K. Kyaw, Yuzhu Chen, and J. Joseph, "Pi-ids: evaluation of open-source intrusion detection systems on Raspberry Pi 2," in *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, Nov 2015, pp. 165–170.

[18] M. V. Mahoney and P. K. Chan, "An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection," in *Recent Advances in Intrusion Detection*, G. Vigna, C. Kruegel, and E. Jonsson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 220–237.

[19] J. McHugh, "Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 4, pp. 262–294, Nov. 2000. [Online]. Available: http://doi.acm.org.eaccess.ub.tum.de/10.1145/382912.382923

[20] C. Osborne, "Meet Torii, a new IoT botnet far more sophisticated than Mirai variants," 2018.

[21] A. Sforzin, F. G. Mármol, M. Conti, and J.-M. Bohli, "Rpids: Raspberry Pi IDS — a fruitful intrusion detection system for IoT," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE, 2016, pp. 440–448.

[22] I. Sharafaldin, A. Habibi Lashkari, and A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," 01 2018, pp. 108–116.

[23] A. Varet and N. Larrieu, "How to generate realistic network traffic?" in *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 299–304.

[24] A. Wright, "Hacking cars," *Commun. ACM*, vol. 54, no. 11, pp. 18–19, Nov. 2011. [Online]. Available: http://doi.acm.org/10.1145/2018396.2018403

Publication VIII

V. Cozzolino, J. Ott, A. Y. Ding and R. Mortier, "ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment," 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), 2020, pp. 223-230, doi: 10.1109/IoTDI49375.2020.00029.

Publication Summary

The development of ubiquitous road-side infrastructure through deployment of stationary and mobile roadside units (RSU) and street furniture seeks out ways to ease congestion and improve road safety. In spite of its great value, smart infrastructure development is still in its infancy with and it is tied to ad-hoc, vertically integrated solutions rather than open platforms offering shared data and compute resources. To support such applications, we proposed a roadside infrastructure which encompasses smart vehicles and devices, RSUs as intermediate computational units, and cloud servers. The challenge then becomes how to enable developers to write and efficiently deploy applications on such a heterogeneous infrastructure. As computing shifts to the edge and particularly the roadside infrastructure, one of the fundamental changes is that it will not be tied to a single vendor.

Hence, we noticed how, from a system perspective, we lack a computational model capable of providing to vehicles reliable and real-time assessment of the road context. To tackle this, we designed ECCO: an orchestration framework that enables edge-cloud collaborative computing for road context assessment. ECCO can create on-demand task execution pipelines spanning multiple, potentially resource-constrained edge-nodes with the smart IoT infrastructure support. Our prototype aimed at creating the groundwork to support new services, which can use more efficiently the road infrastructure and deliver safety-critical applications for road users.

ECCO was developed as an orchestration platform for unikernels: specialised, single address space machine image constructed by using library operating systems. Specifically, our unikernel of choice was MirageOS which we ran on top of the Xen hypervisor.

Author's Contribution

I came up with the idea for the paper as a foundation for an orchestration framework enabling edge-cloud collaborative computing for road context assessment. I have designed, implemented, and evaluated the entire system.

ECCO: Edge-Cloud Chaining and Orchestration Framework for Road Context Assessment

Vittorio Cozzolino, Jörg Ott
Technical University of Munich
{vittorio.cozzolino, ott}@in.tum.de

Aaron Yi Ding
TU Delft
aaron.ding@tudelft.nl

Richard Mortier
University of Cambridge
richard.mortier@cl.cam.ac.uk

***Abstract*—For road safety, detecting and reacting efficiently to road hazards is crucial and yet challenging due to practical restrictions such as limited data availability, which relies on network support. Moreover, from a system perspective we lack a computational model capable of providing to vehicles reliable and real-time assessment of the road context. As autonomous vehicles become widespread, the safety issues are further aggravated by the gap between cloud, roadside infrastructure and road users in terms of communication latency, software-hardware compatibility and data interoperability. To tackle this, we present ECCO: an orchestration framework that enables edge-cloud collaborative computing for road context assessment. ECCO can create on-demand task execution *pipelines* spanning multiple, potentially resource-constrained edge-nodes with the smart IoT infrastructure support. Our prototype lays the groundwork to support new services, which can use more efficiently the road infrastructure and deliver safety-critical applications for road users.**

***Index Terms*—Edge computing, Distributed computing, Unikernel**

I. Introduction

The development of ubiquitous road-side infrastructure through deployment of stationary and mobile roadside units (RSU)[1] [2] and street furniture such as lampposts [3] seeks out ways to ease congestion and improve road safety. For example, detailed metropolitan maps coupled with citywide pollution fingerprinting can improve citizen health, helping pedestrians and cyclists select less polluted routes. In spite of its great value, smart infrastructure development is still in its infancy with and it's tied to ad-hoc, vertically integrated solutions rather than open platforms offering shared data and compute resources. In fact, an open platform supporting multi-tenant access to a citywide compute edge-network infrastructure would facilitate development and deployment of a broad range of applications at a reduced cost.

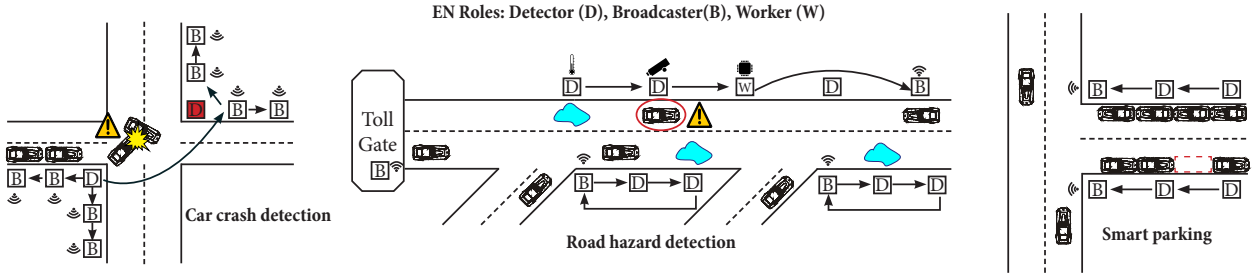
To support such applications, we propose a roadside infrastructure comparable to [4], which encompasses smart vehicles and devices, RSUs as intermediate computational units, and cloud servers. The challenge then becomes how to enable developers to write and efficiently deploy applications on such a heterogeneous infrastructure. As computing shifts to the edge and particularly the roadside infrastructure, one of the fundamental changes is that it will not be tied to a single vendor, regardless of how comprehensive their offerings may be [5]. Hence, solving the problem of balancing and controlling applications deployed by multiple providers is of crucial importance. Moreover, it is not about only edge or cloud — the key for innovation lies in their interplay. In our work, we build on top of these requirements a platform designed to deploy applications instantiated as *edge-cloud pipelines*. Therefore, we design and implement an orchestration framework enabling road context assessment by providing precise information about the road condition. Expanding on our previous work on computation offloading with unikernels [6], we propose a distributed, edge-cloud computational model to deploy multi-node execution pipelines on-demand. Comparing with existing frameworks such as KubeEdge [7] with generic computational model and cloud-only control plane, with ECCO we propose an edge-cloud chaining model dedicated to dynamic IoT scenarios (i.e., road context assessment) and with *responsibility repartition* between cloud and edge.

[1]A Roadside Unit is a V2X direct link transceiver that is mounted along a road or pedestrian passageway [1].

II. Model of Computation

Deployment of the roadside infrastructure poses the non-trivial challenge of assessing the **road context** as the ensemble of *precise* and *trustworthy* road events information, at *scale*. This problem assumes even greater relevance in combination with fully autonomous vehicles, which rely on content delivery through mobile or edge communication to precisely understand the real-time driving environments [8]. With the support of edge computing, we can build an infrastructure able to deliver fresh information to nearby vehicles, enhancing their context awareness. Such approach can enable new services or enhance existing ones such as incident warning broadcasting, traffic signal violation warning, pre-crash sensing, cooperative forward collision warning, lane change warning, black-ice detection.

We can identify static and dynamic entities at work in the roadside scenario which need to communicate and exchange information. Based on these, we devise a model of computation pivoting on three elements: the inputs received by the roadside infrastructure, the functions (*edge functions*, *EF*) processing them, and the outputs enabling different services. To provide a thorough description, we select three use-cases: *(a)* car crash detection, *(b)* road hazards detection, and *(c)* smart parking as shown in Figure 1. The latter illustrates an example of a linear pipeline where: (*i*) the inputs are the



(a) Car crashes on the road can be detected with the support of cameras. The detector node will pass the information to all nearby broadcasters to send a notification to as many vehicles as possible. In this case, a redundant detector node is deployed to counter possible node failures.

(b) Multiple broadcasters notify vehicles entering the highway of the presence of possible hazards ahead of their path. For instance, a car driving against traffic and the presence of black-ice on the road. The former requires a normal camera while the latter is more complex. Photodetectors and thermopiles, infrared cameras (thermography) or Peltier elements installed directly below the asphalt are possible options. In this situations, allocationg different tasks to each detector node is required by their different sensors.

(c) Detectors nodes (D) scan detect the presence of free parking spots directly below them. The detection can be done with cameras, IR proximity sensors or ultrasonic sensors. The acquired information is then disseminated to nearby vehicles by the broadcasting nodes (B).

Fig. 1: Smart roadside infrastructure use-cases. (a) Car crash detection, (b) road hazards detection, (c) smart parking.

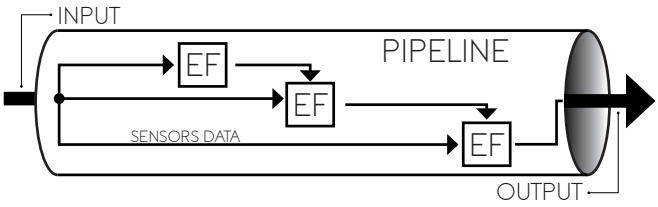


Fig. 2: Visual representation of an ECCO pipeline.

sensors readings detecting free parking spots, (*ii*) the roadside equipment is the compute infrastructure at the edge of network on which the EFs are deployed and executed, and (*iii*) the output is a list of available parking spots which is sent to nearby vehicles. Figure 2 illustrates how these elements are connected to form an ECCO **pipeline**.

We next describe the pipeline components (network, functions, nodes) in more detail, as well as their deployment and execution strategy.

A. Pipeline Components

In our scenario, the vehicles are the recipients of the pipelines output. They receive information from the infrastructure via long-range communication radios such as LoRaWAN [9] or LTE-V2X for Vehicle Fog Computing [10], [11]. As we focus on the computational model and system design, we do not delve deeper into the specifics of V2I transmission mechanisms, a topic explored in other research efforts [12].

Edge Nodes. Following the definition of Shi et al. [13] that edge computing occurs in proximity to datasources, we define an Edge Node (EN) to be a device close to the end-user, such as a mobile phone, PC, or wireless access point. In other contexts, the definition could be extended to include Radio Access Network (RAN) micro-servers [14]. In our case, we focus on the already mentioned RSUs, which are deployed on the road to monitor it and collect data. As they are stationary, we assume good connectivity to the cloud and to other ENs forming what we call an *Edge Network*.

Edge Functions. An Edge Function (EF) is a self-contained, atomic function which embeds a small piece of the application logic that can be executed standalone. When chained together, EFs form an execution pipeline[2]. Each instance of EF plays a specific role and is hosted on a EN. They need to be placed strategically based on the available datasources, the current load status and the geographic position.

Edge-Cloud Pipelines. An ECCO pipeline is a distributed task involving a set of ENs. ENs, listed in the pipeline, take part in execution chains and collaborate to run it. In the next section, more details are provided regarding how such pipelines are deployed.

B. Pipeline Deployment

We envision two levels of control in the pipeline deployment and management process: (*i*) the cloud, which defines the high-level, application driven pipeline deployment plan and (*ii*) the edge which locally makes scheduling decisions based on the parameters described in the rest of this section. The detail of a pipeline structure is defined by the cloud provider, which also monitors its execution.

We assume ENs are reachable from the cloud and can report their available data and current load in terms of active EFs and pipelines. On this basis, the service can plan a pipeline based on a set of parameters to exploit data locality. Once offloaded, the pipeline can be configured to run independently from the cloud, based on specific policies. The need for a constant connection with the cloud stems from the specific scenario. Safety is a major concern in our use-cases as human lives are involved and the constant presence of the cloud as an overseer is deemed necessary to properly manage resources and system failures. For example, dissemination of wrong information or neglecting a car accident may put lives at risk.

As ENs have limited resources and are shared by multiple services, we use the *priority* and *execution* fields in the

[2]The composition of sources (inputs), edge nodes, and sinks (outputs) is similar to a directed acyclic graph (DAG). However, from a user perspective it can be abstracted to a linear flow so that we use the term pipeline to emphasize this relationship.

pipeline configuration to decide when to execute a pipeline. The *priority* field assumes different values based on the use-case and it is static, meaning that a specific use-case will always have the same priority. It is defined by the cloud provider orchestrating the service. For instance, car crash detection will always have higher priority than smart parking. This information allows the system to dynamically shut-down low priority services when additional resources are required by the high priority ones.

Another parameter is *execution*, which can assume only two values: on-demand and automatic. On-demand pipelines are only deployed when requested explicitly by the service provider. Black-ice detection is deployed on-demand as it only manifests in specific conditions (e.g., low temperature at night). Likewise, smart parking is not required in the early hours of the day or when there is very low traffic density detected. Conversely, car crash detection will be flagged as *automatic* as it is a safety critical application running with the highest priority.

Pipelines are flexible and adapt to the use-case and ENs at our disposal. The execution flow can be represented as a directed acyclic graph (DAG) or directed cyclic graphs with topological ordering [15]. We focus on the system aspects as theoretical challenges in service composition techniques have been explored in other studies [16]. For instance, in Figure 1a the pipelines branch to disseminate the alert regarding a car crash as quickly as possible to as many repeater nodes in close proximity. The same behaviour is expected in case of node malfunction, where branching might be necessary to bypass an unresponsive EN. When an EN is not reachable, a substitute is found to replace it or the pipeline is adjusted to skip the node and remove it from the execution tree. For Figure 1a, this means that we will not be able to reach some vehicles directly from our broadcasting ENs if the failure affects a broadcaster node. Conversely, if a detector node goes down, there will be another node ready to replace it and able to detect the car crash. Intersections require redundant ENs deployment as they are often involved in accidents: in 2007, approximately 2.4 million intersection-related crashes occurred, representing 40% of all reported crashes and 21.5% of traffic fatalities [17].

Another reason for branching is that each EN has different resource. One EN might only have cameras, another one only a proximity sensor and a broadcasting interface. This information is collected by the cloud and used to opportunely plan the pipelines structure. ENs without a broadcasting interface can only have a detector role which in turn is defined by its sensors' capabilities. By analogy, there can be ENs playing both the detector and broadcaster role. In the smart parking use-case, the data flow generated by the detectors is progressively enriched along the pipeline. In this case, a small delta of processing is carried out by each detector leaving the broadcasting node only with a task of actually sending the results as shown in Figure 3a. Finally, broadcasting nodes might not support all the radio access technologies required for vehicular communication which is a problem currently discussed by the research community [18].

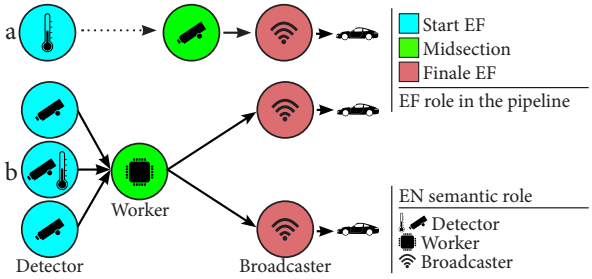


Fig. 3: Pipelines' execution graphs based on ENs capabilities and EFs roles.

In other cases, we might need an additional worker node to perform a computationally intensive task. For example, integration of multiple sensors feeds to detect road hazards as shown in Figure 1b. Another example is an intersection where all the data generated is sent to the worker node, processed, and sent back to manage more efficiently the traffic lights based on the current traffic conditions. The processing node might be a micro-server in close proximity which sends the refined information to selected broadcasting nodes (Figure 3b). The need for multiple broadcasting nodes is twofold: greater communication range and available network interfaces. In fact, radio access technologies required by vehicular communication are changing rapidly and it is expected that not all will be supported by a single RSU [19].

C. Pipeline Execution

The cloud provider generates the pipeline configuration which contains details about the execution plan. When the configuration is offloaded, the ENs involved parse it and each identifies sections it can execute in relation to other nodes. Each pipeline is thus split into sub-pipelines, and transformed into multiple stages which eventually become executable. Execution order of EFs within an EN can be based on various parameters, e.g., priority, expected load, and deadline.

ENs scan the received pipeline configuration and identify the group of EFs it should execute. The classification determines the order to execute and chain EFs, plus the respective roles. An EF has one of three roles: (*i*) *Start*, starting a sequence; followed by (*ii*) zero or more *Midsections*; culminating in (*iii*) a *Finale* which closes the sequence. Sequence ordering parameters are used to correctly unfold execution onto the ENs. The nomenclature adopted in Figure 1 (*detectors*, *broadcasters* and *workers*) applies to the EN while the one just introduced only to the EFs and it is used internally by the system to properly order the pipeline graph. What matters for the pipeline processor is the *relative execution order* of the EFs and not their actual task in relation to the EN capabilities. The relationship between these two concepts is shown in Figure 3 with two simple topologies.

ECCO creates temporary, dynamic execution chains based on the pipeline topology to form ad-hoc collaborative networks of ENs. As data flows from one EF to the next, computation unfolds and progresses toward the Finale. As already

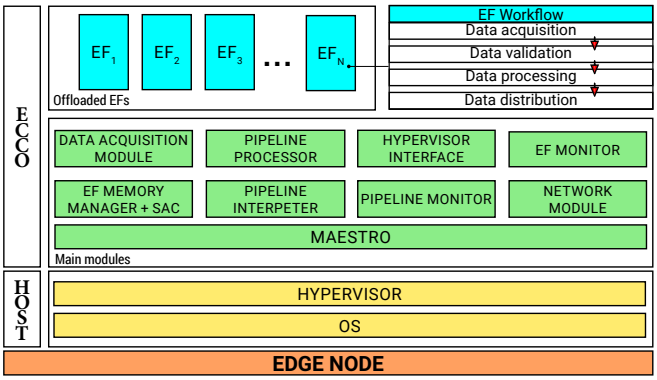


Fig. 4: Overview of ECCO modules.

discussed, pipelines need not be linear but can branch and join to create execution DAGs.

III. ECCO: Design

In this section, we provide an overview of the system depicted in Figure 4 and its components, relate them to our use case, and describe the system workflow. ECCO was designed to achieve two goals: (*i*) provide a landing platform to offload lightweight and fine-grained services orchestrated by the cloud and running on constrained devices at the edge; and (*ii*) support seamless cooperation and interconnection of ENs to support pipelines offloaded from the cloud.

If roadside infrastructure was only usable as an extension of the cloud, reliant on the cloud to work, then a slow or intermittent network connection could render the whole infrastructure useless. Information about road conditions would be retrieved slowly or not at all, and vehicles would be left without information about imminent hazards, potentially costing lives. Treating roadside infrastructure as an edge computing infrastructure, able to use but not reliant upon the cloud, offers a reliable, resilient, and independent infrastructure delivering services even when the cloud is unreachable from end-users.

Crowdsourcing cannot provide this because vehicle density on less heavily used roads will often be insufficient to reliably map road conditions. Available spatial data are sparse and inadequate, leading to incomplete or misleading information distributed to vehicles driving in low-traffic areas. Effectiveness of onboard car sensors is also reduced in common situations as adverse weather conditions which reduce visibility.

ECCO addresses these challenges by providing a platform where multiple cloud services can share existing edge infrastructure for scheduling and handling multiple offloaded pipelines. It offers computational power at the ENs, enabling both independence from the cloud in case of intermittent connectivity, and dynamic processing of information based on chaining EFs.

A. Components

Our system relies on edge offloading: a paradigm that moves computation from the cloud to edge nodes [20]. To differentiate from similar solutions, we design our system as a collaborative framework where multiple ENs are chained to execute different pipelines. To orchestrate the offloaded EFs at the edge, we developed a set of modules running on each EN. The components listed below are associated with the blocks in Figure 4.

Maestro. This is the core of and entry point to our system, functioning as both a coordinator and an interface with the outside world. When one or more EFs are offloaded as part of a pipeline, *maestro* handles the calling of the required modules to filter, order and execute the EFs. During pipeline execution, each EF is tracked and monitored to assess its state in conjunction with the pipeline's. Since multiple parties can access the same ENs, the execution of parallel pipeline is also supported as the allocated resources are completely independent. For resiliency purposes, checkpoints of the pipeline status together with EFs intermediate results are stored in a local database. This modules takes care of bootstrapping ECCO by notifying the presence of an EN to the cloud by advertising its capabilities in terms of hardware resources (e.g., RAM, CPU), sensors, cameras, and communication interfaces. These parameters allow a correct placement of the EFs to minimize distance from the datasource without overloading the EN. In fact, EFs are mapped to ENs based on the required data and type of processing.

EF workflow. Each EF is composed of four phases: data acquisition, validation, processing, and distribution as shown in Figure 4.

In the *data acquisition* phase, an EF awaits the necessary data from the *maestro* which identifies the correct datasource and retrieves the data on the EF behalf. In fact, *maestro* exposes to EFs different end-point to access sensors or local databases identified during the bootstrapping phase. Moreover, the specific steps of the data retrieval phase change depending on the type of end-point. For instance, in the case of hardware sensors, the code to pilot them is embedded directly into the EFs, while for external sources (e.g., databases) *maestro* would use libraries from the host to read the data and then pass it to the EF. Contextualizing, in the example use-case of black ice detection such data are images produced by an infrared camera or readings from a Peltier element. The *data validation* phase checks the received data for errors, eventually requesting a re-transmission. The *data processing* phase is the core of the EF as it contains the developer code. By customizing this part of the EF, it is possible to execute arbitrary code in the EF, granted that eventual external dependencies and libraries have been opportunely handled. In relation to our use-cases, it can contain algorithms to manipulate and process images from cameras or do sensors fusion. Finally, the *data distribution* phase determines whether the outputted result should be passed to the host module which takes care of sending it to nearby road users or sent to the next EF in the local sequence. For instance, a *midsection* EF (detector) in the pipeline can output a post-processed image to be sent along the pipeline for further analysis while a *finale* EF (broadcaster) will signal to nearby vehicles the presence of potential hazards.

Data communication primitives. Dependent on pipeline

structure, data can be exchanged in two ways: (*i*) *Intra-node* communication occurs when the transfer involves two co-located EFs or an EF and the *maestro*; and (*ii*) *Inter-node* communication occurs when the transfer takes place between two EFs on different ENs. To do so, we adopted a shared memory approach to transfer data between EFs using a custom module called *EF memory manager* (EF-MM).

Network module. It enables communication between distinct ENs. It has two groups of queues that contain data structures called *bundles*, a set of parameters to unequivocally identify a pair of producer and consumer ENs. A combination of IDs extracted from the pipeline configuration serves this purpose. The bundles in the inbound queue are stored until consumed by one or more local EFs, which remain on standby until data is available. The outbound queue contains the bundles that are ready to be forwarded to the next EN in the pipeline. The outbound queue is also used as a fail-safe measure in case of a misstep in a pipeline stage whereas data bundles are stored until the malfunctioning nodes are ready to proceed. For added resiliency, the bundles are preserved inside a database to allow hot restart of the system in case of local failure.

Other components. Of the remaining components, the *data acquisition module* is a library wrapper, loaded on-demand based on the requirement specified in the pipeline configuration to interact with different datasources. The *hypervisor interface* exposes an API to control and monitor the VMs running on Xen. The *pipeline processor* contain the core algorithm to unfold the pipeline into ordered sequences. It filters the EFs, identify their roles, bundled them in order sequences (longest sequence). Finally, there is a *pipeline monitor* for each running pipeline as their execution is independent. It spawns multiple *EF monitors* to track each running EF. It uses the *hypervisor interface* to track the status of each running instance and report back in case of failure.

IV. Implementation and Evaluation

ECCO was developed as an orchestration platform for unikernels: specialised, single address space machine image constructed by using library operating systems [21]. Specifically, our unikernel of choice was MirageOS [22] which we ran on top of the Xen hypervisor [23]. We used virtualization to abstract over hardware discrepancies between ENs, and to obtain fine-grained control over running VMs, stronger isolation, and compatibility with existing cloud computing platforms. Our implementation uses C for the EF memory module (a kernel module), Python for the core modules, and OCaml for the EF code (mandated by use of MirageOS).

ECCO computational model is **platform-independent**, meaning that it can be potentially implemented using other sets of technologies as Docker containers on top of Kubernetes. We decided to develop our system based on unikernels due to the multiple advantages in terms of isolation, memory footprint and fine-grained function encapsulation. These are crucial properties in a multi-tenant, resource-constrained scenario.

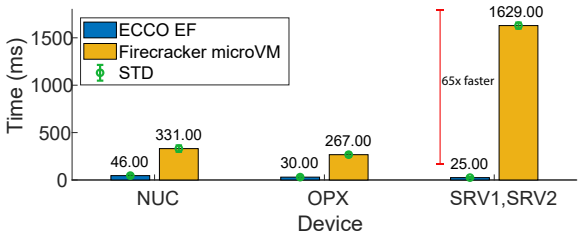


Fig. 5: Average ECCO EF vs. Firecracker microVM boot-up time and standard deviation (STD).

Xen was the most suitable hypervisor for our implementation as it directly supports MirageOS, our chosen unikernel framework due to it producing compact, bootable images that embed only the required OS functionality. Other unikernel technologies were available but MirageOS is one of the most mature and has already been applied to similar IoT use-cases [24]. However, our framework is in principle compatible with any unikernel framework that supports Xen adding flexibility in terms of usable programming languages (e.g., C++, Java, Haskell).

A. Evaluation

Our preliminary evaluation of ECCO focuses on: (*i*) the overhead introduced by the technology choices made in ECCO (Xen, MirageOS) and (*ii*) what is the impact of these overheads on a specific application, driven by our use-cases. The default EF used for in our experiments is a MirageOS unikernel supporting basic image processing operations fed with an image size of approximately 280 kB. This EF is used for all our subsequent benchmarks.

Device	CPU	RAM	Ocaml	Xen	OS
Dell PowerEdge R530 (SRV1, SRV2)	Intel Xeon E5-2640 2.60GHz — 32 Cores	128 GB	4.04.2	4.6.0	Ubuntu 14.04 Kernel 3.19.0
Intel NUC (NUC)	Intel i5-6260U 1.80GHz — 4 Cores	16 GB	4.04.2	4.6.6	Ubuntu 14.04 Kernel 4.4.0
Dell Optiplex 7050 (OPX)	Intel i5-7500T 2.70GHz — 4 Cores	8 GB	4.04.2	4.6.6	Ubuntu 14.04 Kernel 4.4.0

TABLE I: Devices specifications.

Different devices were used to understand the performance gap between the edge and cloud (all connected to the same LAN network). As revealed in Table I, SRV1 and SRV2 have identical configuration, hence their results are bundle together in all the plots due to negligible differences. We performed each experiment 100 times, except when clearly stated.

To evaluate baseline overheads we compare against Amazon Firecracker [25], a recently introduced lightweight serverless computing framework that delivers end-to-end orchestration for tiny VMs. To do so, we built a custom microVM based on an Alpine Linux v3.9 kernel, loaded it with OpenCV v3.4.6 and allocated it was 128 MB RAM and 1 vCPU. The size of its *rootfs* was roughly 4.5 GB.

Figure 5 shows boot times for unikernels compared to the Firecracker microVM. On all devices where we can compare to the Amazon Firecracker microVM, the unikernel

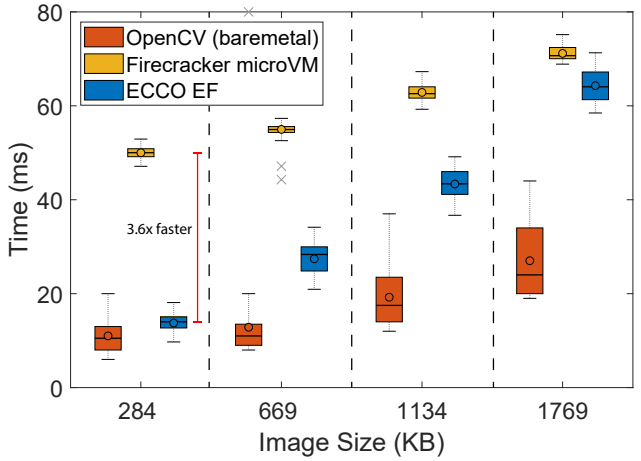


Fig. 6: ECCO Benchmarks (SRV1, color normalization).

boot time was substantially lower, below 50 ms. There is also a considerable difference in size between the images, which probably accounts for much of the difference in boot times. The ECCO EF unikernel is around 5 MB EF while the microVM is around 22 MB plus another 4500 MB of attached *rootfs*. Similarly, the RAM required for an EF is around 15 MB (x86) or 12 MB (ARM) while the microVM required at least 33 MB. We believe that this shows the ECCO approach is well suited to low-latency applications running on ENs with limited memory, as well as for situations where EFs may be updated and distributed frequently.

We compare EF performance against two baselines in Figure 6: (*i*) we developed multiple C++ applications with OpenCV v3.4.6 replicating the operations executed inside the EF; and (*ii*) we loaded the applications in the custom Firecracker microVM previously described. In this way we compared our system to both solutions. For this purpose, we developed a simple application for colour normalization. For space reasons, we present only the result for SRV1, but a similar behavior was shown for the other devices. While ECCO cannot outperform OpenCV running on bare-metal, it has a substantial advantage compared to Firecracker. Pairing this result with the substantial difference in boot time, ECCO outperforms the alternatives and is competitive with bare metal solutions for small image sizes. ECCO performance assumes even greater importance when executing distributed computation spanning multiple ENs, where both quick instantiation and execution time are crucial.

We identify a different execution time growth factors between ECCO and Firecracker, steeper for the former. This shows that our solution is suitable for processing a small amount of information, while the serverless Amazon approach shines with higher data loads.

V. Related Work

Our work draws on multiple strands of existing research which we split into two major branches: detection of road hazards and events, and distributed edge computing systems.

Detecting road conditions and possible hazards is a problem that has been solved in multiple ways: through crowdsourcing, where vehicles exchange collected data to spot bumps [4], or through new infrastructure, where infrared cameras on lampposts are used to identify ice formations on the road [26]. Various studies have examined the efficacy of different methods for detecting road conditions [27], [28].

Current solutions focus on using either crowdsourcing or edge networks for transferring road conditions information. However, the quality of crowdsourced spatial data is often unreliable [29], resulting in insufficient density of data to estimate road conditions in low-traffic areas. Solutions based on on-board car sensors can prove to be mediocre depending on the road characteristics and weather conditions. Edge computing can play a pivotal role in addressing these challenges by exploiting road infrastructure to augment vehicle sensory capacity beyond their on-board sensors. Using edge computing to support offload of computation to deliver particular applications is not new [13]. With ECCO we are concerned with providing a distributed framework to dynamically interconnect nodes based on the applications requirements.

Numerous authors have explored offloading computation and data, for different purposes and under different decision policies [30]–[34]. Cloudlets [35] were a particular pioneer in the field of computation offloading. Earlier work from Madhavapeddy et al. [36] proposed on-demand specialized VM instantiation within connection setup time. Airbox [37] presents a software platform based on onloading and backend-driven cyberforaging. It shares the general direction presented in our paper in terms of offloading the EF. Compared with Airbox, ECCO achieves fine-grained offloading by using unikernels instead of Docker technology. Databox [38] proposes a hybrid physical and cloud-hosted system for personal data management. Koller et al., [39] also proposed an unikernel-based serverless framework architecture while a more recent research effort proposes a WebAssembly solution [40].

VI. Conclusions

ECCO is a distributed edge computing framework developed to deliver road context assessment. We discuss the advantages of our approach in comparison to crowdsourcing, cloud computing, and onboard car sensors solutions. Given the fast adoption of autonomous vehicles, our work propose a computational model to bridge the gap between cloud, road infrastructure and road users to deliver rapidly instantiated, on-demand services. The logic, design and implementation of our system were described in relation to the analyzed scenario and encompass two crucial problems of edge computing: fine-grained orchestration and collaborative, multi-device task execution at the edge. At the core of our framework, the function chaining allows different nodes to cooperate in the execution of ECCO pipelines.

References

[1] "US government publishing office (2017) electronic code of federal regulations, 47 cfr part 90." http://www.ecfr.gov/cgi-

bin/textidx?tpl=/ecfrbrowse/Title47/47cfr90-main-02.tpl, 2017, [Online; accessed 07-October-2019].

[2] "V2X OBU deployed in new york's cv pilot," https://www.traffictechnologytoday.com/news/autonomous-vehicles/v2x-obu-deployed-in-new-yorks-cv-pilot.html, 2019, [Online; accessed 07-October-2019].

[3] "Smarter togheter - Munich's smart lamp posts shine," https://www.smarter-together.eu/news/munichs-smart-lamp-posts-shine, 2018, [Online; accessed 07-January-2019].

[4] S. Basudan, X. Lin, and K. Sankaranarayanan, "A privacy-preserving vehicular crowdsensing-based road surface condition monitoring system using fog computing," *IEEE Internet of Things Journal*, vol. 4, no. 3, pp. 772–782, 2017.

[5] "Satya Nadella looks to the future with edge computing," https://techcrunch.com/2019/10/08/satya-nadella-looks-to-the-future-with-edge-computing/, 2019, [Online; accessed 09-October-2019].

[6] V. Cozzolino, A. Y. Ding, and J. Ott, "Fades: Fine-grained edge offloading with unikernels," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM, 2017, pp. 36–41.

[7] "Kubeedge," https://kubernetes.io/blog/2019/03/19/kubeedge-k8s-based-edge-intro/, 2019, [Online; accessed 21-October-2019].

[8] Q. Yuan, H. Zhou, J. Li, Z. Liu, F. Yang, and X. S. Shen, "Toward efficient content delivery for automated driving services: An edge computing solution," *IEEE Network*, vol. 32, no. 1, pp. 80–86, 2018.

[9] A. J. Wixted, P. Kinnaird, H. Larijani, A. Tait, A. Ahmadinia, and N. Strachan, "Evaluation of LoRa and LoRaWAN for wireless sensor networks," in *2016 IEEE SENSORS*. IEEE, 2016, pp. 1–3.

[10] X. Hou, Y. Li, M. Chen, D. Wu, D. Jin, and S. Chen, "Vehicular fog computing: A viewpoint of vehicles as the infrastructures," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 6, pp. 3860–3873, 2016.

[11] Y. J. Li, "An overview of the DSRC/WAVE technology," in *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer, 2010, pp. 544–558.

[12] K. Abboud, H. A. Omar, and W. Zhuang, "Interworking of DSRC and cellular network technologies for V2X communications: A survey," *IEEE transactions on vehicular technology*, vol. 65, no. 12, pp. 9457–9470, 2016.

[13] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[14] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[15] Wikipedia contributors, "Topological sorting — Wikipedia, the free encyclopedia," 2020, [Online; accessed 31-January-2020]. [Online]. Available: "https://en.wikipedia.org/wiki/Topological_sorting"

[16] U. Sadiq, M. Kumar, A. Passarella, and M. Conti, "Service composition in opportunistic networks: A load and mobility aware solution," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2308–2322, 2014.

[17] "The national intersection safety problem," https://bit.ly/2pFzaUe, 2009, [Online; accessed 21-October-2019].

[18] L. GomesBaltar, M. Mucck, and D. Sabella, "Heterogeneous vehicular communications-multi-standard solutions to enable interoperability," in *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2018, pp. 1–6.

[19] G. Naik, B. Choudhury, and J.-M. Park, "Ieee 802.11 bd & 5g nr V2X: Evolution of radio access technologies for V2X communications," *IEEE Access*, 2019.

[20] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan 2018.

[21] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," *Queue*, vol. 11, no. 11, pp. 30:30–30:44, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2557963.2566628

[22] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *Acm Sigplan Notices*, vol. 48, no. 4, pp. 461–472, 2013.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[24] J. Zhao, T. Tiplea, R. Mortier, J. Crowcroft, and L. Wang, "Data analytics service composition and deployment on iot devices," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: ACM, 2018, pp. 502–504. [Online]. Available: http://doi.acm.org/10.1145/3210240.3223570

[25] Amazon, "Firecracker – Lightweight Virtualization for Serverless Computing," 2020, [Online; accessed 31-January-2020]. [Online]. Available: "https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/"

[26] M. Kutila, M. Jokela, J. Burgoa, A. Barsi, T. Lovas, and S. Zangherati, "Optical roadstate monitoring for infrastructure-side co-operative traffic safety systems," in *Intelligent Vehicles Symposium, 2008 IEEE*. IEEE, 2008, pp. 620–625.

[27] M. Jokela, M. Kutila, and L. Le, "Road condition monitoring system based on a stereo camera," in *Intelligent Computer Communication and Processing, 2009. ICCP 2009. IEEE 5th International Conference on*. IEEE, 2009, pp. 423–428.

[28] Y. Iwasaki, M. Misumi, and T. Nakamiya, "Robust vehicle detection under various environmental conditions using an infrared thermal camera and its application to road traffic flow monitoring," *Sensors*, vol. 13, no. 6, pp. 7756–7773, 2013.

[29] A. Comber, L. See, S. Fritz, M. Van der Velde, C. Perger, and G. Foody, "Using control data to determine the reliability of volunteered geographic information about land cover," *International Journal of Applied Earth Observation and Geoinformation*, vol. 23, pp. 37–48, 2013.

[30] A. Y. Ding, B. Han, Y. Xiao, P. Hui, A. Srinivasan, M. Kojo, and S. Tarkoma, "Enabling energy-aware collaborative mobile data offloading for smartphones," in *2013 IEEE International Conference on Sensing, Communications and Networking (SECON)*. IEEE, 2013, pp. 487–495.

[31] E. Hyytiä, T. Spyropoulos, and J. Ott, "Offload (only) the right jobs: Robust offloading using the markov decision processes," in *Proceedings of IEEE WoWMoM '15*, 2015.

[32] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE Infocom*. IEEE, 2012, pp. 945–953.

[33] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[34] D. F. Willis, A. Dasgupta, and S. Banerjee, "Paradrop: A multi-tenant platform for dynamically installed third party services on home gateways," in *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, ser. DCC '14. New York, NY, USA: ACM, 2014, pp. 43–44. [Online]. Available: http://doi.acm.org/10.1145/2627566.2627583

[35] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.

[36] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam *et al.*, "Jitsu: Just-in-time summoning of unikernels," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 559–573.

[37] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using airbox," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 14–27.

[38] R. Mortier, J. Zhao, J. Crowcroft, L. Wang, Q. Li, H. Haddadi, Y. Amar, A. Crabtree, J. Colley, T. Lodge *et al.*, "Personal data management with the databox: What's inside the box?" in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*. ACM, 2016, pp. 49–54.

[39] R. Koller and D. Williams, "Will serverless end the dominance of linux in the cloud?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 169–173. [Online]. Available: https://doi.org/10.1145/3102980.3103008

[40] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19.

New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084