Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Sistemi di Elaborazione delle Informazioni

Self-Management of Geographically Distributed Infrastructures and Services

Doctoral Dissertation of:
Danilo Filgueira Mendonça

Supervisor:
Prof. Luciano Baresi

Tutor:
Prof. Raffaela Mirandola

The Chair of the Doctoral Program:
Prof. Barbara Pernici

2014/2 – XXX' Cycle

First and foremost, I wish to thank CNPq —National Council for Scientific and Technological Development— for the crucial financial support provided to Brazilian students who adventure abroad. I extend my gratitude to the Brazilian taxpayers who made this possible.

My most sincere gratitude and appreciation for the fellow PhD colleagues who shared the struggle and joy of the process. Special thanks to the people who I had the opportunity to learn, share and develop ideas. Above all, I (still) believe in collaboration over competition.

To Dr Fernando Antonio Reis Filgueira, professor, researcher, writer, philosopher, and grandfather. Your lessons will never be forgotten.

Throughout this challenging path, we often find ourselves devoid of light, self-esteem, or will to overcome the challenge. I can only attempt to express my gratitude for all the steady dedication, companionship, encouragement and love received from my friend, girlfriend, and now wife, Tábata. Together we go further.

To my beloved family and friends in Brazil, my gratitude for your loyal transatlantic support.

Last but not least, I wish to thank my adviser Luciano Baresi for his mentorship and valuable feedback on the various steps and skills required to fulfil the goals of my PhD.

Abstract

THE paradigm of edge computing emerged in the last decade aiming to fill the gap between cloud data centres— accessible through multiple hops of networking — and the *prosumers* of information residing at the network edge. In edge computing, computing and storage resources are co-located with different kinds of infrastructures: with cellular infrastructures like base stations and aggregation sites; with core network components like ISP gateways; and with private infrastructures. Among the main goals of edge computing are the mitigation of network delay and the increase of bandwidth required by latency-sensitive and data-intensive applications hosted on mobile and Internet of Things (IoT) devices, including Autonomous Vehicles, Augmented/Virtual Reality, Mobile Multi-player Games, Natural Language Processing, Real-time Data Analytics, and Industry 4.0.

The decentralised nature of the edge computing paradigm entails many challenges. First and foremost, the fine-grained distribution of edge nodes impose limitations to the capabilities offered by each node. Resources must be managed efficiently, and collaboration among surrogate nodes is paramount to enable more customers and services to be admitted into the system. Operational (Ops) aspects like the placement, deployment and scaling of edge-based services make automation and self-management properties first-class requirements. Last but not least, existing application and service models need to be adapted to cope with the characteristics of densely distributed infrastructure: heavyweight, monolithic applications may not fit into edge node resources or may fail to scale.

In this thesis, we tackle the materialisation of edge computing from two main perspectives: architectural, in which we focus on the application and service models that enable the offloading of computation from latency-sensitive and data-intensive applications to edge nodes; and management, which handles the autonomic configuration, deployment, and scaling of services by geo-distributed infrastructures.

At the heart of our proposal is the paradigm of serverless computing and the Function-as-a-Service model. We leverage this alternative approach to cloud computing and propose a Serverless Architecture for Multi-Access Edge Computing. We then expand our contribution landscape with heterogeneous resources from mobile, edge, and cloud platforms —which we refer to as the Mobile-Edge-Cloud Continuum. To tackle the life-cycle of serverless functions deployed to the Continuum, we propose A3-E framework. A3-E moves away from centralised orchestration and management in favour of opportunistic, autonomic and decentralised provisioning of Function-as-a-Service to mobile applications with distinct requirements such as latency and battery consumption. We conclude our contributions with PAPS, a comprehensive framework that tackles the effective and efficient placement and scaling of serverless functions onto densely distributed edge nodes through multi-level self-management and control theory.

Sommario

Il paradigma del *edge computing* è emerso nell'ultimo decennio al fine di colmare il divario tra i data center cloud —accessibile attraverso vari hop di networking— e i consumatori e produttori di informazioni che risiedono ai margini della rete. In edge computing, le risorse informatiche sono co-localizzate con diversi tipi di infrastrutture: stazioni di radio base e siti di aggregazione; con componenti della rete backbone come ISP gateways; e con infrastrutture di rete private. Tra gli obiettivi principali del edge computing vi sono la riduzione del ritardo di rete e l'aumento della larghezza di banda richiesta dalle applicazioni sensibili alla latenza e throughput ospitate su dispositivi mobili e IoT, compresi veicoli autonomi, realtà aumentata / virtuale, giochi multiplayer mobile, elaborazione del linguaggio naturale, analisi dei dati in tempo reale e Industry 4.0.

La natura decentralizzata del paradigma del edge computing comporta molte sfide. Innanzitutto, la distribuzione a grana fine dei nodi edge impone limitazioni alle capacità offerte da ciascun nodo. Le risorse devono essere gestite in modo efficiente e la collaborazione tra i nodi più vicini è fondamentale per consentire a più clienti e più servizi di essere ammessi nel sistema. Aspetti operativi (Ops) come il posizionamento, il deployment e il ridimensionamento dei servizi edge rendono le proprietà di automazione e autogestione requisiti di prim'ordine. Infine, i modelli di cloud esistenti devono essere adattati per far fronte alle caratteristiche di un'infrastruttura densamente distribuita: le applicazioni pesanti e monolitiche potrebbero non adattarsi alle risorse dei nodi edge o potrebbero non riuscire a scalare.

In questa tesi, affrontiamo la materializzazione del edge computing da

due punti di vista principali: architettonico, in cui ci concentriamo sui modelli di applicazione e di servizio che consentono lo scarico del calcolo da applicazioni sensibili alla latenza e ad alta intensità di dati ai nodi edge; e gestione, che si occupa della configurazione, l'implementazione e il ridimensionamento autonomi dei servizi da parte di infrastrutture geo-distribuite.

Il cuore della nostra proposta è il paradigma del serverless computing e del modello Function-as-a-Service. Sfruttiamo questo approccio alternativo al cloud computing e proponiamo un'architettura serverless per Multi-Access Edge Computing. Quindi espandiamo il nostro panorama di contributi con risorse eterogenee da piattaforme mobili, edge e cloud — che chiamiamo Mobile-Edge-Cloud Continuum. Per affrontare il ciclo di vita delle funzioni serverless distribuite al Continuum, proponiamo un framework A3-E. A3-E si allontana dall'orchestrazione e dalla gestione centralizzata in favore di un provisioning opportunistico, autonomo e decentralizzato di Function-as-a-Service per le applicazioni mobili con requisiti distinti quali latenza e consumo della batteria. Concludiamo il nostro contributo con PAPS, un framework multi-livello che affronta il posizionamento e il ridimensionamento efficace ed efficiente delle funzioni serverless su nodi edge densamente distribuiti attraverso la teoria dell'autogestione e del controllo.

Contents

1 Introduction **1**
1.1 The Emergence of Edge Computing 1
1.2 Research Opportunities and Challenges 3
1.2.1 Architectural Aspects 3
1.2.2 Management Aspects 4
1.3 Problem Statement and Research Goals 5
1.4 Contributions 7
1.5 Dissemination 8
1.6 Structure of the Thesis 9

2 Preliminaries **11**
2.1 Mobile Computing and the Internet of Things 11
2.2 Cloud Computing 13
2.2.1 Virtualisation Technologies 13
2.2.2 Cloud Service Models 16
2.3 Edge Computing and Similar Concepts 17
2.3.1 Early History 18
2.3.2 Cloudlets 19
2.3.3 Mobile Ad-Hoc Clouds 19
2.3.4 Multi-access Edge Computing 20
2.3.5 Fog Computing 22
2.3.6 Summary 23
2.4 Serverless Computing 24
2.4.1 Function-as-a-Service 24

2.4.2 FaaS vs Typical Cloud Service Models 26
2.4.3 FaaS vs Microservices 28
2.5 Autonomic Computing 31
2.5.1 Adaptation Taxonomy 33
2.5.2 Multiple-Criteria Decision-Making 35

3 A Serverless Architecture for Multi-Access Edge Computing 37
3.1 Overview 37
3.2 MEC Architecture and Use Cases 38
3.2.1 ETSI Framework and Reference Architecture for MEC 39
3.2.2 MEC Use Cases 40
3.2.3 MEC Requirements 41
3.2.4 Running Example 42
3.3 The Serverless MEC Architecture 46
3.3.1 Self-Managed Computing Services 46
3.3.2 Supporting Services and Optimisations 49
3.4 The Serverless MEC Platform 57
3.4.1 Platform Architecture 57
3.4.2 Platform Deployment 63
3.5 Proactive Recovery Protocol 65
3.5.1 Overview 65
3.5.2 Platform States 65
3.5.3 Proactive Recovery Bounds 66
3.5.4 Recovery in Action 73

4 Mobile-Edge-Cloud Continuum Through Serverless and Autonomic Computing 75
4.1 Overview 75
4.2 The Mobile-Edge-Cloud Continuum 76
4.2.1 Running Example 78
4.3 System Model 79
4.3.1 Infrastructure Model 79
4.3.2 Continuum Functions and Requirements 81
4.3.3 Continuum Application 82
4.3.4 Mobile Middleware and Domain Manager 83
4.3.5 Life-cycle Management Problem 85
4.4 A3-E Framework 87
4.4.1 Overview 87
4.4.2 A3-E's Awareness 87
4.4.3 A3-E's Acquisition 90

4.4.4 A3-E's Allocation 92
4.4.5 A3-E's Engagement 94
4.4.6 Running Example 95
4.5 Domain Manager 96
4.5.1 Architecture Overview 98
4.5.2 Awareness Manager 98
4.5.3 Acquisition Manager 99
4.5.4 Allocation Manager 102
4.5.5 Engagement 106
4.6 Mobile Middleware 106
4.6.1 Architecture Overview 106
4.6.2 Continuum Application Registration 108
4.6.3 Awareness Manager 109
4.6.4 Acquisition Manager 113
4.6.5 Allocation Manager 114
4.6.6 Library Proxy 119
4.6.7 Mobile Domain 120

5 The PAPS Framework **121**
5.1 Overview 121
5.2 Management Challenges 122
5.3 System Model 123
5.3.1 Infrastructure Model 123
5.3.2 Function-as-a-Service 124
5.3.3 Management Problem Formulation 125
5.4 The PAPS Framework 127
5.4.1 System-Level Self-Management 127
5.4.2 Community-Level Self-Management 128
5.4.3 Node-Level Self-Management 134
5.5 PAPS Simulator 136
5.5.1 PeerSim 137
5.5.2 Implementation Overview 137

6 Evaluation **141**
6.1 Serverless MEC Architecture 141
6.1.1 Overview 141
6.1.2 Application Scenario: Mobile Augmented Reality . . 141
6.2 Experimental Evaluation 144
6.2.1 Goal-Question-Metric 144
6.2.2 Experimental Setup 147

6.2.3 Results: Memory Footprint . . . 148
6.2.4 Results: Overhead and Response Time . . . 149
6.2.5 Results: Simultaneous Users and Function Entropy . 153
6.2.6 Results: Elasticity . . . 158
6.2.7 Discussion . . . 159
6.2.8 Threats to Validity . . . 160
6.3 A3-E Framework . . . 162
6.3.1 Experimental Setup . . . 162
6.3.2 Response Time and Scalability . . . 163
6.3.3 Battery Consumption and Execution Time . . . 165
6.3.4 Domain Selection and Availability . . . 167
6.3.5 Enorm . . . 168
6.3.6 Threats to Validity . . . 170
6.4 PAPS Framework . . . 172
6.4.1 Experimental Setup . . . 172
6.4.2 Partitioning . . . 172
6.4.3 Allocation, Placement and Scaling . . . 173

7 Related Work **177**
7.1 Serverless MEC Architecture . . . 177
7.2 A3-E Framework . . . 179
7.3 PAPS Framework . . . 186

8 Conclusions and Future Work **193**
8.1 Future Work . . . 195

Bibliography **197**

CHAPTER 1

Introduction

1.1 The Emergence of Edge Computing

In the last decade, cloud computing has become a successful model for the rapid provisioning of computing resources on demand. Cloud vendors benefit from the economy of scale by aggregating a pool of configurable and virtually unlimited resources into data centres. Among its variants, the *Infrastructure-as-a-Service* (IaaS) model —empowered by virtualisation technologies— allows users to deploy and scale applications on *virtual machines* [137] and, more recently, *containers* [80].

The advent of *mobile computing* and the *Internet of Things* (IoT) poses new challenges to the centralised data centre model. Data-intensive applications enabled by mobile and IoT devices lying at the network edge are already pushing the boundaries of how much data needs to be transported and processed by cloud data centres [18]. Current estimations point out the continuous and exponential increase in this trend, as more devices and applications join the network. For instance, according to Cisco [101], the total amount of data created (and not necessarily stored) by any device will reach *847 zettabyte* (10^{21}) per year by 2021. Simultaneously, the number and diversity of applications with compute-intensive features

are also increasing, which partially nullify the gains in computing power and battery life achieved in modern pervasive devices. To mitigate battery drain, these applications need to offload compute-intensive tasks to more powerful servers in the cloud, which became known as *Mobile Cloud Computing* [115]. However, real-time applications with strict requirements for latency and throughput raise further questions about the feasibility of the data centre model in satisfying all these needs [92, 97].

The paradigm of edge computing aims to tackle the challenges above by filling the gap between cloud data centres and data prosumers at the network edge [16, 108]. Since the concept of cloudlets [97] was initially proposed by Satyanarayanan and his colleagues, edge computing has been associated with different terminologies, architectures and technologies, all of which share the common goal of enabling new types and scales of interactive, real-time, and data-intensive applications to operate at the network edge. Amongst prominent infrastructure-centric models are Multi-Access Edge Computing (previously known as Mobile Edge Computing) [100]; follow-me cloud [113]; mobile edge clouds [125]; and fog computing [77].

Some of the prominent use cases target by edge computing are consumer-driven. In this category are new and disruptive applications such as Mobile Augmented Reality, Cognitive Assistance, Connected and Autonomous Vehicles, Industry 4.0, to name a few [16, 18, 37]. Network operators and third-party are also expected to harness the power of decentralised infrastructure to deliver new types of services, e.g. to smart city applications that require the processing and analysis of massive amounts of data [103]. In this thesis, we distinguish two main use case categories: *latency-sensitive computation offloading* and *data pre-processing*. As the name suggests, the former considers latency as a first-class requirement. Its primary goal is to *augment the capabilities of mobile and IoT devices*. In contrast, the second scenario is mainly concerned with the anticipation of the processing of voluminous data near its source, often IoT devices.

The satisfaction of the aforementioned use cases created new research opportunities in multiple fields and areas ranging from computer sciences to networking and telecommunications.

From the communication and architectural perspectives, several works tackled the deployment of IT resources in the context of multi-access edge computing and fog computing [61, 64]. Many authors focused on the decision of if and when to offload tasks to edge nodes [78, 119], whereas others tackled the management of edge nodes and the allocation of resources [71, 121]. The dynamic placement and migration of services [62, 125] and the scheduling of tasks [46, 130] also received considerable attention from the

operational research and optimisation communities.

Although some authors leverage the collaboration among end-user devices to achieve the previously mentioned goals [27,116,133], we opted for concentrating our research effort on infrastructure-centric edge computing. The rationale behind this decision is twofold. First, we consider that both models —device-centric and infrastructure-centric— to be complementary and likely to co-exist. In some cases, stable and reliable servers might be key for the creation of device-to-device cooperation among volatile devices. Secondly, we argue that each approach poses significant challenges of its own, which could not be properly addressed within the scope of this thesis.

1.2 Research Opportunities and Challenges

1.2.1 Architectural Aspects

Edge computing has been contemplated with substantial research effort and contributions. However, some important gaps remain. From the software architecture perspective, the particularities of the decentralised infrastructure model envisioned by edge computing are often ignored. The same is valid for the service model to be adopted by edge infrastructure providers. For example, many works refer to *applications*, *components*, and *services* without entering into details about their architecture. In some cases, typical cloud service models are adopted, e.g. virtual machines hosting monolithic applications are provisioned on demand.

Although edge-centric architectures are broad enough to encompass infrastructures ranging from regional data centres [56] to single-board computers [17], it is unlikely that edge nodes will match the capabilities of nowadays' data centres. The geographic dispersion of servers entails additional challenges to the replication of consolidated cloud computing models and software architectures. Thus, a more precise definition of the service model provided by edge infrastructure operators and the architectural model of consumer applications is needed.

Following the breakthroughs in visualisation technologies, the paradigm of *Serverless Computing* [38,58,87] has been conceived to allow developers to focus on the core aspects of their application, while a third-party cloud provider handles the management of the infrastructure required for its execution. To the present date, this paradigm has been developing in different forms and more prominently as stateless compute runtime services, also known as the *Function-as-a-Service* model.

In the context of edge computing infrastructure, serverless computing has critical advantages over other cloud service models. First, functions

are lightweight and stateless by design. Not only functions are good candidates to fit into the limited computing and storage resources, but stateless components can also be scaled and managed more efficiently. This is particularly true in a geographically distributed environment. Secondly, the delegation of infrastructure management to a single entity (the provider) not only allows more efficient provisioning and allocation of resources but also facilitates the configuration of individual servers and the deployment of software components across a potentially large number of servers.

Despite its advantages, serverless computing is still young if compared to other consolidated architectures and models. The adoption of serverless computing and the Function-as-a-Service model in the fulfilment of edge computing use cases target by this thesis needs further assessment. First, the benefits and limitations of modelling, implementing and deploying application logic as functions must be evaluated. In a similar direction, key performance indicators such as *resource allocation efficiency*, *latency overhead* and *scalability* of existing serverless platforms must be assessed and potentially optimised to fulfil the requirements at hand.

1.2.2 Management Aspects

In a typical cloud computing deployment, users define the policies that will govern the provisioning and allocation of resources to their applications [60]. In most cases, a policy defines the conditions for triggering the provisioning (scale-out) or releasing (scale-in) of virtual machines and, more recently, of containers. Accordingly, users have the control and responsibility for determining how much resources are demanded from providers. Such a high autonomy level is possible, thanks to the virtually unlimited resources that characterise nowadays' cloud data centres.

The management of geographically distributed infrastructures and services entails multiple challenges. First, the decentralised nature of edge computing adds a spatial dimension to the resource allocation and provisioning problems. Not only the amount of resources procured to edge-based applications needs to be dynamically decided, but also the placement of these applications (e.g. single or multiple components) onto the mesh of geo-distributed servers. Second, if we consider the limitation of resources at each node, user-centred scaling decisions are likely to conflict with each other unless arbitrated by the infrastructure provider.

For instance, centralised management approaches usually rely on a global system view [91]. Depending on the scale and complexity, a massive amount of data regarding the availability of resources and the performance of hun-

dreds or thousands of components at multiple locations needs to be monitored and analysed. Optimal solutions may be impractical, as the solution space grows exponentially with the number of nodes [122].Management decisions such as service placement and scaling must be consistent with the current system state and therefore to be taken and executed in a timely manner—at the risk of being innocuous or yielding adverse effects. Hence, the delay introduced by networking and complex analysis may undermine the effectiveness and efficiency of the management process.

In the opposite direction, fully decentralised management solutions like those based on the principles of self-organising systems are known for their superior responsiveness and scalability. However, the emergence of behaviour [23] characterising these systems may be hard to analyse and anticipate. Furthermore, decisions based solely on local knowledge may perform poorly from a global perspective. The consequence can be the inability of anticipating adverse or even catastrophic situations, as well as the inefficient distribution of resources among various applications and services.

Following a cloud-like approach, many works tackled the problems above by assuming edge resources to be always available [71,135]. Others do consider capacity limitations [114, 136], but still rely on single-level virtualisation (i.e., virtual machines) instead of more resource-efficient containers. Few proposals tackle the management of containerised applications while considering the resource limitations of edge nodes [28]. To the best of our knowledge, comprehensive edge-centric solutions based on the paradigm of serverless computing are still missing.

1.3 Problem Statement and Research Goals

In light of the aforementioned considerations and challenges, the first research goal addressed by this thesis can be stated as follows:

> *To study, analyse, and evaluate the techniques, models and technologies for the provisioning of computing services and propose an efficient and scalable approach that enables the offloading of computation and anticipation of data processing by compute-intensive, latency-sensitive, and data-intensive applications.*

Due to fundamental aspects of serverless computing, we decided to deepen the investigation of this particular paradigm and to focus on the Function-as-a-Service model. The benefits and limitations of this choice must, therefore, be analysed and evaluated regarding the application scenarios this thesis aims to address, namely the *offloading of latency-sensitive*

computation and the *anticipation of data processing*. Accordingly, the overall research goal can be decomposed into two research sub-goals:

Research goal 1 (RG1) – Architecture

> "*To study, analyse, and evaluate the Function-as-a-Service model and associated technologies; and to propose a solution for the provisioning of efficient and scalable Computing Services to be hosted by geographically distributed infrastructures (edge computing) in the context of compute-intensive, latency-sensitive and data-intensive applications hosted by mobile and IoT devices.*"

First and foremost, RG1 aims to study, adapt and improve the *Function-as-a-Service* model from both theoretical and technological points of view, taking into account the advantages and shortcomings of the infrastructure model characterising edge computing and the specific requirements from the targeted application scenarios.

The proper dimensioning of application components and the assignment of responsibilities in terms of state and behaviour are critical to the materialisation of an edge-cloud ecosystem; not only it must cope with application requirements, but it must also be aligned with the challenges previously discussed. Accordingly, from the architectural point of view, RG1 also concerns: (i) the definition of which application components should be deployed to *edge servers*, to *cloud data centres*, or to remain part of *front-end applications* hosted by mobile and IoT devices; and (ii) the interaction among each of these parts.

Research goal 2 (RG2) – Self-Management

Once we have assessed the Function-as-a-Service model and the proposed architecture, we considered the problem of managing edge systems resulting from the combination of densely distributed, heterogeneous infrastructures providing the services and resources needed for the execution of functions with different requirements. Due to its intrinsic complexity, the management of the resulting distributed system is defined as an *autonomic computing* [47, 112] problem. More specifically, the *self-management* of decentralised infrastructures and services implementing a serverless architecture must cope with: (i) the intrinsic challenges of the infrastructure model; (ii) the requirements of each targeted application scenario; and (iii) the particularities of the Function-as-a-Service model. Accordingly, our second research goal is defined as follows:

"*To develop an effective and efficient solution for the self-management of geographically distributed infrastructures (edge computing) providing computing services for the execution of serverless functions (Function-as-a-Service) in the context of compute-intensive, latency-sensitive and data-intensive applications hosted by mobile and IoT devices.* "

1.4 Contributions

In this section, we outline the contributions of the thesis, mapping them to the research goals stated above.

Research Goal 1 — Architecture

We have investigated and compared different service models and cutting edge virtualisation technologies for the provisioning of computing services by decentralised infrastructures. We also evaluated key aspects regarding the architecture of applications from our targeted use cases. Based on our findings, we proposed a Serverless Architecture for Multi-Access Edge Computing along with the building block components, services, and optimisations of a platform implementing the proposed architecture. The proposal was evaluated both qualitatively and quantitatively through relevant application scenarios and experiments.

Research Goal 2 — Self-Management

Once the benefits and limitations of the proposed architecture had been demonstrated, we expanded the landscape of our contributions with a broader range of deployment configurations resulting from the combination of mobile, edge, and cloud computing. The resulting Compute Continuum extends the Serverless MEC Architecture. To manage the life-cycle of serverless functions deployed to the heterogeneous infrastructures comprising the Continuum, we proposed A3-E framework.

A3-E framework exploits both serverless and autonomic computing to allow serverless functions to be opportunistically and autonomously fetched, deployed and exposed as services by distinct edge and cloud providers, or consumed locally. A3-E prototypes were implemented and evaluated. Results demonstrated significant gains in terms of response time and battery consumption, which are critical requirements for the application scenarios targeted by this thesis.

The opportunistic nature of A3-E yields an efficient allocation of resources. Nonetheless, A3-E does not tackle the collaboration among edge

nodes from a single edge provider. We address this limitation with PAPS, a framework designed to tackle the self-management of densely distributed edge nodes. PAPS is based on a multi-level, decentralised self-management approach that partitions the larger scale edge topology into delay-aware communities and allocates resources among and within communities.

Differently from the optimisation frameworks found in the literature, the PAPS framework combines optimal allocation and placement at the community level with a fast node-level container scaling to render the overall self-management process both efficient and effective. Our framework also comprises a simulator platform, which enables the evaluation of different infrastructure, application, and workload scenarios. The PAPS framework was evaluated with a large scale Multi-Access Edge Computing topology hosting up to 100 serverless functions. Results showed the criticality of the node-level self-management to achieve a more efficient allocation of resources and prevent SLA violations.

1.5 Dissemination

The research conducted during the PhD program has lead to a number of publications. This section lists them in order in which they are presented in the thesis. The list also includes a minor research topic publication.

Major Research Topic Publications

- Luciano Baresi, Sam Guinea, Danilo Filgueira Mendonça. A3Droid: A framework for developing distributed crowdsensing. *In Proceedings of the International Conference on Pervasive Computing and Communication Workshops*. IEEE, 2016, pp. 1-6.

  While this workshop paper [11] pre-dates our research effort with decentralised infrastructures and services, it falls in the category of device-to-device edge computing. Some of the limitations in this paper inspired us towards the infrastructure-based solutions presented in this thesis.

- Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga. Empowering low-latency applications through a serverless edge computing architecture. *In Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing* (ESOCC 2017). Springer, 2017, pp. 196-210.

This conference paper [12] is the basis of Chapter 3, where we introduce the serverless architecture for Multi-Access Edge Computing. It is also the baseline for the Mobile-Cloud-Continuum model presented in Chapter 4.

- Luciano Baresi, Danilo Filgueira Mendonça. Towards a Serverless Platform for Edge Computing. *In Proceedings of the 1st International Conference on Fog Computing* (TO APPEAR). IEEE, 2019.

  This conference paper [14] is the basis for the optimisations and additional services provided by the Serverless MEC Platform presented in Chapter 3.

- Luciano Baresi, Danilo Filgueira Mendonça, Martin Garriga, Sam Guinea, and Giovanni Quattrocchi. A Unified Model for the Mobile-Edge-Cloud Continuum. ACM Transactions on Internet Technology, 19(2):29:1–29:21, April 2019.

  This journal paper [13] is the basis of Chapter 4, where we present the Mobile-Edge-Cloud Continuum model and A3-E, a framework for tackling the management of the life-cycle of functions opportunistically deployed to the Mobile-Edge-Cloud Continuum.

Minor Research Topic Publications

- Danilo Filgueira Mendonça, Genaína Nunes Rodrigues, Raian Ali, Vander Alves, Luciano Baresi. GODA: A goal-oriented requirements engineering framework for runtime dependability analysis. *Information and Software Technology*, 80:245 - 264, 2016.

  This journal paper [65] is the result of previous research effort on goal-oriented dependability analysis with parametric probabilistic model checking. As future work, we envision the extension of this work for enabling dependability analysis of FaaS workflows.

1.6 Structure of the Thesis

The structure of this thesis is as follows:

- Chapter 1 overviews the problem statement and contributions in this thesis, while Chapter 2 provides background information on the theory, techniques, and technologies that are either used by the contributions in this thesis or are central to their understanding.

- Chapter 3 introduces the Serverless Architecture for Multi-Access Edge Computing.
- Chapter 4 introduces the Mobile-Edge-Cloud Continuum and the A3-E framework along with prototype implementations.
- Chapter 5 introduces the PAPS framework along with the simulation platform accompanying the framework.
- Chapter 6 is divided in three parts. Section 6.1 presents a comprehensive evaluation of the Serverless MEC Architecture, whereas Sections 6.3 and 6.4 report on the evaluation of A3-E and PAPS frameworks respectively.
- Chapter 7 surveys related work in the context of edge-centric architectures and the management of geographically distributed infrastructures and services.
- Finally, Chapter 8 provides some concluding remarks, discusses limitations of the proposed approaches and how these limitations can be addressed as part of future work.

CHAPTER 2

Preliminaries

This chapter provides background information on the theory, techniques and technologies that are fundamental for the contributions in this thesis, or central to their understanding.

Section 2.1 briefly covers the current state of the technologies encompassing *mobile computing* and the *Internet of Things*, which are the main drivers for edge computing. In Section 2.2, we discuss relevant aspects of the paradigm of *cloud computing*. Also in this section are presented the consolidated and cutting-edge virtualisation technologies and discuss their role in different cloud service models. In turn, Section 2.3 provides a broad overview of the *edge computing* paradigm. Next, Section 2.4 introduces the serverless computing paradigm and the Function-as-a-Service model. Finally, Section 2.5 introduces the field of *Autonomic Computing*, on which are based our contributions regarding the self-management of decentralised infrastructures and services.

2.1 Mobile Computing and the Internet of Things

The massive popularisation of smartphones and other mobile devices has been accompanied by an increasing number of new and sometimes disrup-

tive applications. The dynamic and interactive nature of many applications demands that data be distributed over the Internet through heterogeneous networks. To fulfil these requirements, mobile applications are typically designed using the client-server model. In this kind of architecture, part of the application logic and state, known as the *client* or *front-end* application, is hosted by devices themselves. In turn, components forming the *back-end* application are hosted by remote servers.

While computational power and battery autonomy significantly increased over the years, mobile devices are not yet able to handle all sorts of tasks [95]. Moreover, the excessive use of mobile applications (often battery hungry) is nowadays considered as a pathological behaviour [3] —one that affects not only the life of its users but also the battery of their devices.

The offloading of computation from mobile devices to cloud servers was first addressed by the concept of *Mobile Cloud Computing* [115]. Its primary purpose was to enable rich applications to run in resource-constrained mobile devices by delegating compute-intensive tasks to cloud servers. However, this approach is limited by network latency, which is prohibitive for most real-time and interactive applications.

In a parallel, but closely related thread, the *Internet of Things* (IoT) [35] have expanded the pervasive computing landscape with a variety of physical devices and everyday objects distinguished by their capabilities of sensing and actuating the environment, and of communicating and interacting over the Internet [2]. Examples include but are not limited to *Autonomous and Connected Vehicles*, *Smart Homes*, *Smart Cities*, *Smart Industry* (also known as Industry 4.0), and different sorts of wearable technology. Contrasting with smartphones, tablets and other mobile devices, IoT is generally specialised and lack the resources for performing more sophisticated computation; instead, connectivity is harnessed to allow data to be collected and processed elsewhere —in most of the cases, by cloud-based services.

Mobile and IoT devices have in common the role of data prosumers, i.e., they are both consumers and producers of data. According to predictions, these data will skyrocket in the coming years [101], mainly driven by mobile and IoT devices —from simple temperature sensors to highly sophisticated Autonomous/Connected Vehicles, passing by all sorts of wearable devices [18]. Although cloud computing appears like a straightforward solution for processing such massive amounts of data, the latency introduced by long back and forth transmissions can severely hurt user experience. Further, it would entail a considerable burden to the network infrastructure, which may fail to deliver the required Quality-of-Service.

2.2 Cloud Computing

The last decades witnessed the consolidation of cloud computing [5, 137] as a practical solution to manage and leverage IT resources and services. Cloud computing reduces initial investments (i.e., no need to buy servers) and maintenance costs. Instead, resources or service applications are offered as (remote) services, available *on-demand* and *on-the-fly*, and billed according to a *pay-as-you-go* model.

While cloud computing also comprises private deployments (also known as the private cloud), it is mainly associated with the public data centre model. In public cloud, shared pools of configurable computer system resources and higher-level services owned by a third-party can be rapidly provisioned with minimal management effort, often over the Internet. Empowered by cutting-edge technologies, cloud data centres are known for giving end-users —from garage practitioners to large-scale conglomerates— access to virtually infinite resources.

Within this section, we provide an overview of the leading virtualisation technologies used in cloud computing. We give particular attention to operating system-level virtualisation (containerisation), an enabler technology for serverless computing and architecture. This section concludes with a brief introduction to the leading cloud service models. In Section 2.4, we shall present a more detailed comparison between these models and the emerging Function-as-a-Service model.

2.2.1 Virtualisation Technologies

Virtualisation technologies are paramount to the success of cloud computing [137]. IBM introduced the concept of virtualisation more than fifty years ago to allow multiple applications to run simultaneously. Since then, the term evolved, and nowadays virtualisation is applied in many areas of computing, including computer hardware, storage and network resources.

In cloud computing, virtualisation is mainly used as a mean to (logically) separate the server infrastructure and make it available to many users (also referred to as *tenants*) in an isolated fashion. Leading virtualisation technologies can be categorised into two levels: *hardware level virtualisation*, which includes *full virtualisation* and *paravirtualisation*, and *operating-system-level virtualisation* (also known as *containers*) [80, 90].

Hardware level virtualisation

In full virtualisation, every salient feature of the hardware is reflected in *virtual machines* (VM). Hence, a VM provides the same functionality as a

physical computer. In other words, they offer an *efficient, isolated duplicate of a real computer machine*. The latter is commonly referred to as the *host* and the VM as the *guest*. To achieve full virtualisation, a *hypervisor* uses native execution to share and manage hardware, allowing for multiple environments which are isolated from one another, yet exist on the same physical machine. In *hardware-assisted* full virtualisation, the hypervisor relies on hardware capabilities, primarily from the host processors, to bypass the binary translation

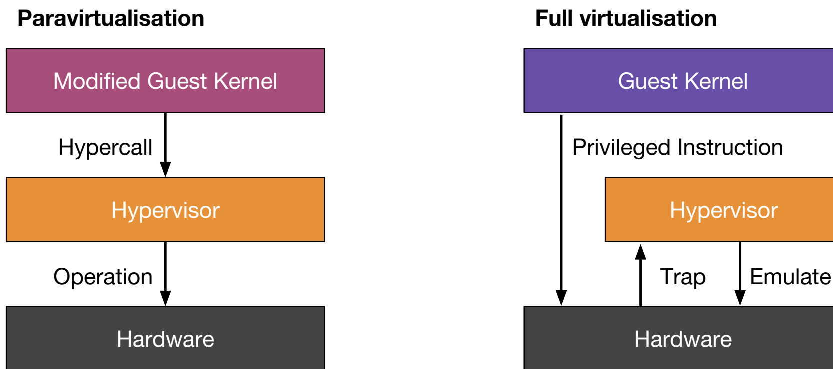


Figure 2.1: *Comparison between para and full virtualisation approaches. Paravirtualisation requires the modification of the guest kernel, which becomes "paravirtualisation aware" and makes use of special-purpose API.*

In paravirtualisation, guest VMs do not attempt to issue commands to the (simulated) hardware, but instead makes use of drivers to issue commands directly to the host operating system (see Figure 2.1). Accordingly, the guest operating system must be *paravirtualization-aware* (i.e., to be explicitly ported to para-API) to run on top of a paravirtualisation hypervisor.

Operating system-level virtualisation

More recently, operating-system-level virtualisation (or containerization) gained significant attention. This virtualisation approaches exploits features of an operating system in which multiple isolated user-space instances share the same kernel [29, 110]. In contrast with VMs (see Figure 2.2), a container consists of an isolated process that runs directly on the operating system without the intermediation of a hypervisor. For the program(s) running in them, containers mimic real computers. Contrary to a regular program running in a physical machine, containerised software are restricted in their view and access to the operating system's and underlying (virtual)

machine's resources (e.g. access to memory, CPU, and storage). Similarly to VMs, containers are organised as *images*.

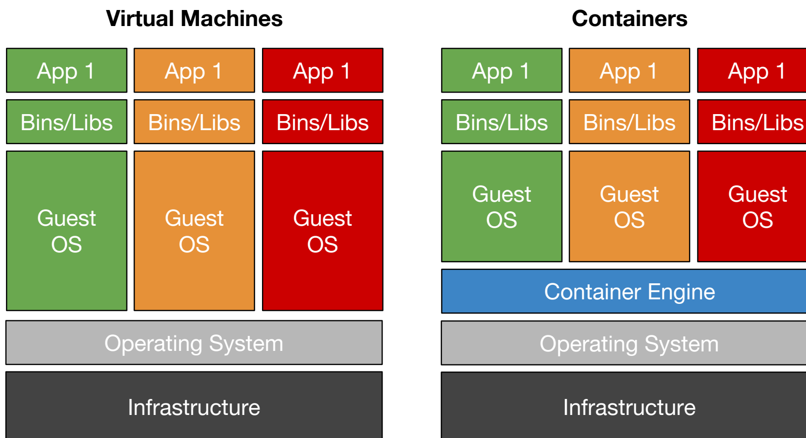


Figure 2.2: *Comparison between virtual machines and containers*[1]

Containers have many advantages compared to virtual machines [110], namely:

- **Overhead:** unlike other virtualisation technologies, there is very little or no overhead since it uses the host operating system kernel for execution, i.e., they do not require the intermediation of a hypervisor nor custom APIs. Depending on the implementation, containers impose little or no overhead.
- **Image size:** containers are considerably lighter since they do not contain an operating system kernel, but only the application and user space binaries and libraries. A container image can be as small as a few megabytes. For instance, the size of a container image based on a minimal Linux distribution (Alpine Linux) is 5MB only[2]. In contrast, virtual machines images comprise all of the parts of an operating system. While VM images based on minimal Linux distributions are much lighter (order of a hundred of megabytes) than more traditional distributions (order of gigabyte), it is still significantly more onerous than its container counterpart.
- **Start time:** the time needed to start a container is similar to the start time of a native process. Conversely, a virtual machine start time comprises the booting time of a complete operating system. Hence, a VM

[2] https://hub.docker.com/_/alpine

start time is usually orders of magnitude higher (minutes [63]) than the time to start a container (seconds).

There are also disadvantages to using containers that must be considered. In terms of security, containers provide a lower degree of isolation as they share the same kernel —often with root access. The kernel takes care of isolating containers, e.g. through virtual-memory support. Hence, security threats have easier access to the entire system when compared with hypervisor-based virtualisation. Another drawback of containerization is the reduced OS flexibility (OS lock-in). While distinct guest OS can live side by side on the same host, a single OS must be used by all containers. Also, container images are not compatible across different OS.

2.2.2 Cloud Service Models

Cloud providers offer resources and services at three different layers: at the *Software-as-a-Service* (SaaS) layer, users can remotely access full-fledged software applications; at the *Platform-as-a Service* (PaaS) layer, one finds a development platform, a deployment and a runtime execution environment, which is used to run user-provided code in sandboxes hosted on cloud-based premises; at the *Infrastructure-as-a-Service* (IaaS) the user can access computing resources such as virtual machines, block storage, firewalls, load balancers, or networking I/O. Next, we briefly introduce the more relevant service models in the context of our work.

Infrastructure-as-a-Service

In its most popular form, IaaS consumers rent virtual machines to run their (guest) software programs. In this way, many users can share the same physical machine without knowing it. The National Institute of Standards and Technology (NIST) defines IaaS as:

> "*The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer can deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).*"

[2] Adapted from https://blog.netapp.com/blogs/containers-vs-vms/

While containers are considered as a type of IaaS, for the sake of clarity and presentation we opt for distinguishing container-based infrastructure provisioning into a separate category, namely *Container-as-a-Service*; from now on, the term IaaS refers exclusively to VM-based provisioning.

Platform-as-a-Service

A PaaS provider typically offers developers a set of services that allow them to develop, test, deploy and run applications without having to worry about the underlying infrastructure. According to NIST, PaaS is defined as:

> "*The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.*"

Container-as-a-Service

Recent developments of the IaaS model empowered by the container technology lead to a new type of cloud service, namely the *Container-as-a-Service* (CaaS). This model exploits the advantages of operating system-level virtualisation in the provisioning of elastic compute services. While the unique characteristics of CaaS position it between IaaS and PaaS, it is most commonly viewed as a subset of IaaS [80].

The high granularity and simplicity of containers demand the support of managing tools. For instance, an application may comprise hundreds or more containers running on a server. Following the widespread adoption of this virtualisation approach, various technologies are nowadays available and under active development, including security tracking, monitoring, orchestration, and schedules that oversee operations. Hence, major CaaS providers offer additional services to mitigate the complexity of container management.

2.3 Edge Computing and Similar Concepts

This section summarises the paradigm of edge computing in its different shapes and flavours, including background information about its early history and some of the conceptual and architectural variations in the literature.

2.3.1 Early History

In recent years the paradigm of edge computing has received significant attention from academia and industry. Edge computing has roots in the content delivery network (CDN), introduced by Akamai in the 1990s [96]. Back then, latency mitigation was the main driver for the distribution of computing and storage resources to the network edge. Edge computing extends CDN by allowing not only the caching and distribution of multimedia data but also the execution of general purpose computing at the network edge.

Still in the 1990s, experiments have demonstrated the feasibility of implementing speech recognition on a resource-constrained mobile device by offloading computation to a nearby server [95]. Similar experiments demonstrated savings in battery life. Already in the 2000s, Satyanarayanan et al. generalised these concepts using the term *cyber foraging*. However, in this decade, much of the attention from both industry and research turned to the emerging paradigm of cloud computing.

Despite the consolidation of cloud computing and its success in providing virtually infinite compute and storage resources, the latency and jitter resulting from multiple network hops between mobile devices and cloud datacenters motivated the resurgence of edge computing. In a 2009 article, Satyanarayanan et al. [97] defined *cloudlets* as trusted, resource-rich computer or cluster of computers accessible through WiFi and available for use by nearby mobile devices. Proximity and mobile computation offloading are the two fundamental characteristics of the cloudlets architecture.

Fog computing [18] was later on introduced by Bonomi and his colleagues from Cisco and referred to a *dispersed cloud infrastructure*. Its primary purpose is to address the scalability of IoT infrastructure, as the volume of data generated by existing and emerging applications is expected to increase at exponential rates [101]. In contrast with cloudlets, fog computing reference architecture [77] consists of a multi-tier, hierarchical deployment of fog nodes co-located with heterogeneous public and private infrastructures stretching from cloud data centres to densely distributed nodes at the network edge.

Last but not least, multi-access edge computing [100], formerly referred to as mobile edge computing, was conceived as a network architecture concept that enables computing capabilities and IT services at the edge of the cellular network. MEC differentiates from cloudlets and fog computing by co-locating compute and storage resources within or in proximity to the cellular network infrastructure, which allows traditional mobile and new IoT

devices to leverage edge services while connected to the mobile network.

Next, we further described each of the aforementioned edge computing strands.

2.3.2 Cloudlets

Cloudlets [97] —trusted, resource-rich computer or cluster of computers— were proposed to be co-located with WiFi access points in private (e.g., domestic) or public (e.g., hot spots) deployments. Its original purpose was to enable resource-poor mobile devices to perform compute-intensive and energy-hungry tasks (e.g., those from mobile applications augmenting human cognition).

The dependence with WiFi is considered the main drawback of cloudlets. Nowadays, mobile devices enjoy pervasive connectivity through mobile networks. In contrast, the coverage and support for mobility provided by WiFi access points are limited. Notwithstanding this, the number of public WiFi hotspots in urban areas is already substantial and continues to increase. Furthermore, cloudlets are suitable for various application scenarios associated with specific locations that are likely to be covered by WiFi and that users are expected to remain stationary for a while (e.g., domestic, office, industry and public locations).

2.3.3 Mobile Ad-Hoc Clouds

Also referred to as mobile edge-clouds, mobile device clouds, or simply mobile cloud, the concept of mobile ad-hoc clouds consist of another option for performing computing at the edge [27, 116, 133]. In this approach, groups of mobile devices, often with some common objective to achieve, form ad-hoc clouds in which their resources are shared and combined. The ultimate goal is to process high demanding applications locally.

The formation of ad-hoc clouds entails several critical challenges. First, devices in proximity must be found while guaranteeing that processed data will be delivered back to the source. Second, computing devices must coordinate their activities despite the lack of control channels providing reliable communication. Third, device owners must be motivated to provide the computing power of their devices, given the battery consumption and additional data transmission constraints. Last but not least, the creation of ad-hoc clouds may incur in various security issues.

2.3.4 Multi-access Edge Computing

The multi-access edge computing [100] was conceived as a network architecture concept that enables computing capabilities and IT services at the edge of the cellular network. Also in this case, the baseline is that running applications and services closer to end-users provides the benefit of reduced latency, jitter, and higher throughput.

The most significant difference between MEC and the strands of edge computing introduced before relies on its integration with cellular network infrastructure. MEC combines elements from both information technology (IT) and modern telecommunications. In one hand, services deployed at the MEC can benefit from the context information regarding the client's location and what kind of user equipment is connected to the base station. In the other, mobile network operators may leverage context information to adapt the traffic and avoid congestions at the radio and backhaul network [61].

In the literature, multiple variations of the MEC architecture have been proposed. Next, we briefly present the most relevant models.

Small cell cloud (SCC)

The *Small cell cloud* (SCC) [59] aims to enhance small cells (base stations) such as microcells, picocells and femtocells with computing and storage resources. It is argued that, because a large number of such cells is supposed to integrate future mobile networks, the SCC can provide enough computational power for edge services that have strict requirements on latency.

The SCC architecture specifies the role of a small cell manager (SCM). The latter is responsible for performing dynamic and elastic management of virtualised resources from a single SCC cell, with a local SCM deployed within the RAN; or multiple SCC cells forming a cluster, with a centralised SCM at a higher cellular infrastructure hierarchy. SCMs can also form a hierarchy of control, with local SCMs being managed by remote SCMs. The latter can decide where to deploy and migrate a given computation within a cluster in order to optimise the service delivery for the end-user.

Mobile micro cloud (MMC)

Similarly to the SCC, the mobile micro cloud (MMC) [123] also defines an architecture in which cells are enhanced with computing and storage resources. Differently from the SCC architecture, MMC resources are not clustered but dedicated to the clients connected to its base station. Accordingly, the management of resources forms each MMC is fully distributed

and decentralised. Nonetheless, MMCs are interconnected to guarantee service continuity through virtual machine migration whenever clients move from one base station to another.

Fast moving personal cloud (MobiScud)

The fast moving personal cloud (MobiScud) [120] exploits the new paradigms of software defined networks (SDN) and network function virtualisation (NFV) to integrate cloud services into mobile networks without disrupting backward compatibility with the existing mobile networks. Differently from SCC and MMC architectures, MobiScud envisions the placement of computing and storage resources not precisely at the cells, but within or close to the RAN. As such, it is regarded as a decentralised cloud architecture. MobiScud also includes a manager component responsible for interfacing with the SDN switches and the decentralised cloud. Its purpose is to monitor user equipment activities (e.g., handover), to orchestrate the allocation of resources and the routing of data traffic within the SDN.

Follow me cloud (FMC)

Similarly to the MobiScud, the Follow-Me Cloud (FMC) proposes an architecture in which services deployed to distributed datacenters follow its mobile clients as they roam through the mobile network. Contrasting to previous approaches, the FMC places distributed datacenters further from users, into the core network of the mobile service operator. Each data centre is statically or dynamically mapped to base stations based on different metrics. A manager entity is responsible for the resources and services and decides which datacenters should serve which clients. The latter may be deployed either centrally, hierarchically, or in a fully distributed configuration, similarly to the SCC architecture.

CONCERT

The main difference from CONCERT and the previous architectures relies on the hierarchical distribution of computing and storage resources within the mobile network. Additionally, CONCERT integrates with centralised cloud datacenters. Its purpose is to flexibly and elastically manage resources, including the network. In CONCERT, local servers with lower computational power offer low latency, lightweight services, whereas more resourceful regional and centralised data centres serve as a fall-back in case local resources are not sufficient and as providers of additional services with higher latency. CONCERT also distinguishes between control

and data planes. The control entity (conductor) may also be deployed in a centrally or hierarchical manner, whereas the data plane is composed of communication, computing, and storage resources presented as virtual resources. The latter is monitored and managed by the control entity.

ETSI MEC

The ETSI can be considered as the dominant player within the MEC community. Among the white papers and technical specifications regarding the MEC paradigm and architecture, ETSI specified a first reference architecture [105] for MEC. It addresses multiple aspects, including functional elements — software entities that run on top of virtualised resources from one or more MEC servers — and reference points allowing the interaction among them.

ETSI's Reference Architecture comprises different deployment schemes. In the first, MEC servers are co-located with base stations, similarly to the SCC and MMC architectures. Alternatively, MEC services can be placed at aggregation sites or multi-RAT aggregation points. Finally, MEC servers can reside farther from the user equipment at some higher level in the mobile networks infrastructure: for instance, at the edge of the core network, analogously to the FMC architecture.

As one of the most prominent types of edge computing deployment, we have adopted ETSI's MEC framework and architecture as the baseline for our contribution regarding our first research goal. Accordingly, we shall provide further details on the ETSI's MEC Framework and Reference Architecture in Chapter 3.

2.3.5 Fog Computing

In contrast with the concept of cloudlets, fog computing does not limit its architecture to servers lying a few hops away from its users. Instead, fog computing comprises a multilevel hierarchy of nodes stretching from cloud data centres to densely distributed servers at the network edge. While cloudlets and consumer-centred MEC services are mostly concerned with the offloading of latency-sensitive computation, fog computing gives special emphases to the anticipation of processing and analysis of exponentially larger volumes of data generated and consumed by IoT applications. Hence, in addition to latency, fog computing considers network bandwidth as a critical resource, especially for enabling data-intensive IoT use cases like Autonomous and Connected Vehicles. Notwithstanding their differences, fog and edge computing have in common the goal of filling the gap

between cloud data centres and applications at the network edge.

In 2015, several companies, including Cisco, Microsoft, Intel, and Dell, in partnership with Princeton University, created the OpenFog consortium. Among others, this consortium produced the first reference architecture for fog computing [77]. It has also been actively involved in the dissemination and development of the edge computing paradigm with the publication of both technical and academic papers.

In our work, we have used both ETSI's and OpenFog's reference architectures. More precisely, ETSI's specifications were the principal source for the initial development of an architecture targeting latency-sensitive applications. We find ETSI's particularly well detailed in terms of the components of a MEC Platform. In contrast, we leveraged the broader deployment perspective associated with fog computing and detailed in OpenFog's Reference Architecture, which is less detailed in terms of functional components and their interactions. In particular, the N-tier fog deployment has been used as a reference for the contribution targeting the self-management of geographically-distributed infrastructures and services in Chapter 5.

2.3.6 Summary

The architectural variations of MEC and edge computing in general not only verse on how to deploy and manage edge resources, but also on the nature of applications and services that are expected to benefit from such resources. In one hand, finely grained and fully distributed edge nodes, physically located at cellular base stations (as proposed by SCC and MMC), are expected to deliver the network performance required by real-time and interactive applications. In the other hand, these nodes may fail to provide the resources needed by heavyweight applications and services.

To cope with distinct needs, a comprehensive architecture as proposed by CONCERT seems more realistic. It also leverages the potential of conventional cloud computing and supports the dynamic allocation of virtual resources from local, regional, and cloud data centres. Thus, the work presented in this thesis adopts CONCERT's vision and complements it with the following characteristics:

- Multiple edge and cloud providers are expected to co-exist, enabling the application to decide which provider to use whenever such choice applies.
- In addition to on-demand, cloud-like provisioning of edge services, we assume the existence of cloudlets accessible through WiFi hotspots.

> The latter represent several application scenarios involving mobile computing and the Internet of Things, including smart home, smart city, and Industry 4.0. Hence, we assume that applications will be able to seamlessly harnesses various types of edge-centric infrastructures.

Throughout this thesis, the term *edge* is used in reference to the paradigm itself, or to the extremity of the network. The latter becomes more evident in the context of an N-tier fog deployment, which distinguishes three levels: the edge level, the fog level, and the cloud level. More details on the N-tier fog deployment are presented in Chapter 5.

2.4 Serverless Computing

Recently, serverless computing [31, 87, 131] emerged as a compelling alternative for hosting applications in the cloud. The essential characteristic of the serverless computing paradigm consists of the delegation of the management of infrastructure to one or more service providers. Serverless providers become responsible for the elastic provisioning of resources, networking, load balancing, fault tolerance, and for the patching and configuration of servers.

Serverless computing and architectures are associated with two concepts [87]: of applications that rely on third-party cloud services for handling business logic and state (also known as *Backend-as-a-Service*, or *BaaS*); and of applications for which server-side logic is still written by application developers, but, unlike traditional architectures, functions run in stateless compute containers that are event-triggered and fully managed by a third party (also known as *Function-as-a-Service*, or *FaaS*).

2.4.1 Function-as-a-Service

A Note of Advise

Leading vendors like Amazon's AWS Lambda, Google's Cloud Functions and Microsoft's Azure Functions do not disclose details on their respective serverless solutions. As shown in different studies [8, 58], the *Quality-of-Service* may vary considerably from one FaaS platform to the other. Behind these discrepancies are the distinct policies and mechanisms governing the provisioning and allocation of operator's resources [8, 58].

In this section, we aim to introduce the FaaS model, as advertised or documented by the leading commercial and open source solutions. Whenever appropriate, we shall refer to specific platforms when describing the key features of this emergent service model.

Programming Model

In FaaS, application logic is implemented as stateless functions. A function consists of a granular logic unit responsible for a specific task. This characteristic enables a separation of concerns and reinforces modular development, making the code more maintainable and easily modifiable.

The essential capability of a FaaS platform is that of an event processing system [9]. Functions are executed in reaction to various sorts of events. The majority of FaaS platforms allow functions to be triggered remotely, often through a RESTful API. Additionally, commercial platforms usually provide integration with other services (e.g. a notification service) that may be used as *event source*. Functions can be triggered either synchronously (also known as *blocking invocation*) or asynchronously [3].

Functions may be written in various programming languages, including both interpreted and compiled ones. Most platforms [4, 34, 67] support a wide variety of languages like Java, Python, JavaScript, PHP, and Ruby. Through accessible extension mechanisms, open source platforms such as OpenWhisk [104] and OpenFaaS [106] further extend this support to any language that may run in a containerised runtime.

Execution Model

Upon function activation, a *containerized compute runtime* (CCR) is either made available (*cold* container) or reused from previous execution (*warm* container) by a *dispatcher*. Additionally, the virtual machine(s) hosting function containers are provisioned on demand (VM cold) or reused. While booting a VM usually take minutes [58], CCRs are initialised within milliseconds or a few seconds, thanks to the container technology. Nonetheless, retaining containers *warm* for a while prevents the CCR initialisation overhead (also known as *cold start*) for subsequent requests.

The period in which VMs and CCRs are kept warm varies according to the platform and vendor [58]. For instance, Apache OpenWhisk [104] —a leading open source FaaS platform— retains warm containers for 60 seconds in its default configuration. Similarly to AWS Lambda and other platforms, OpenWhisk's commercial counterpart (namely IBM's Cloud Functions) make use of specific policies and optimisations in the dynamic provisioning and deprecation of virtual machines and containers.

[3] In the remainder of this work, we shall refer to *function activation* when the sort of event that triggers its execution is not distinguished; to *function invocation*, when it is triggered synchronously; and to *function request*, when it is triggered (synchronously or not) by an external source.

Packaging and Deployment

Functions are deployed to the FaaS platform as a *self-contained source code file* or as a *package* that main contain dependencies (e.g., software modules and libraries, multimedia assets, and other assets). Functions are uniquely identified with a *name* within the tenant's *namespace*. In most platforms, functions may be additionally specified with:

- **Compute Runtime**: specifies what *runtime* should be used for executing the function (e.g. `Python:3`, `NodeJS:6`, `Java:1.8`). It may additionally refer to native/third-party binaries and libraries used by the function (e.g. `ImageMagick`, `OpenCV`, to name a few).
- **Memory**: specifies the maximum memory that can be allocated for the runtime; often passed as `memory` parameter to the *Docker Engine*.
- **Timeout**: defines the time limit after which the function is terminated by the platform with an error.
- **Concurrent Executions**: defines the limit of simultaneous executions after which new activations are *throttled* by the platform.

Most FaaS-based platforms also enforce overall limitations to the function *package size* and *execution time* by design. These limits are essential for keeping the time needed for loading functions into containers adequately low, and to render the allocation and provisioning of resources more predictable and therefore efficient [9].

2.4.2 FaaS vs Typical Cloud Service Models

FaaS vs IaaS

The IaaS model provides users with access to voluminous cloud resources. Resource elasticity is managed at the virtual machine level, often resulting in over-provisioning of resources leading to increased hosting costs, or under-provisioning that results in poor application performance. Such disparity is usually worsened by abrupt workload variations.

Figure 2.3 illustrates the problem above with a typical IaaS deployment subject to an inconsistent traffic pattern. Bursts of workload (in blue) are seen as spikes that, within a time window, reach only twice the full capacity of the allocated virtual machines (in red). Most of the time, the workload is constrained to a small fraction of the allocated capacity, resulting in under-utilization. Despite the improvements in full virtualisation technologies,

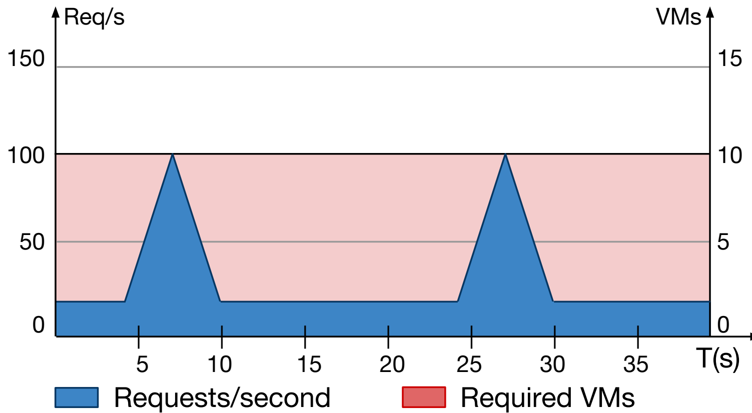


Figure 2.3: *Inconsistent traffic pattern in a typical IaaS deployment [87]*

virtual machines still require a significant amount of time to boot, which impedes responding to abrupt workload fluctuations [10].

Contrasting with the previous scenario, the FaaS model allocates resources (i.e., containers) when actually needed. While the use of virtual machines enables applications to share physical resources, the FaaS model leverages the container technology to deliver the following advantages:

- A single operating system is shared among concurring functions serving one or multiple applications.
- Functions are promptly deployed and scaled in reaction to events without pre-allocating computational resources.

FaaS vs CaaS

This CaaS model exploits the advantages of operating-system-level virtualisation in the provisioning of elastic compute services. Although CaaS shares with FaaS many of the benefits of container technology, these models are fundamentally different. First, CaaS is still considered as an infrastructure abstraction. Similarly to IaaS, consumers are still in charge of management decisions (e.g. defining rules for auto-scaling). Second, it is the responsibility of the user to define what goes in a container, including versions of the containerised software and underlying runtime. Finally, users are billed per time of allocated container and not just for their actual usage.

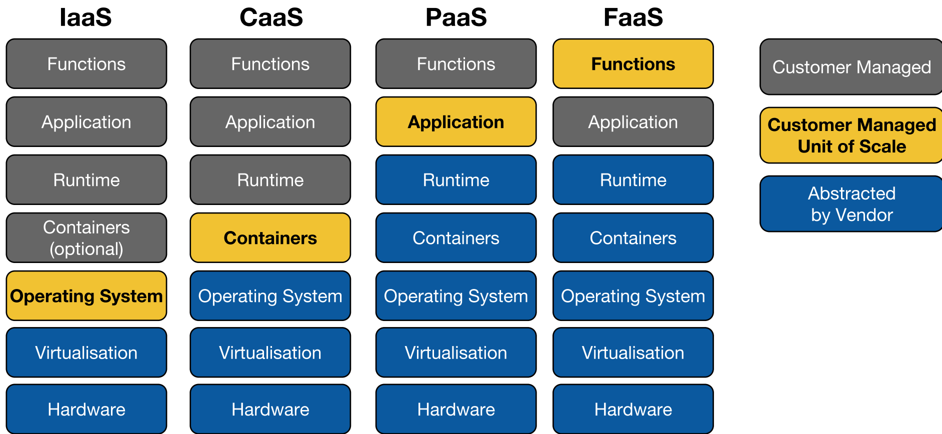


Figure 2.4: *Comparison between traditional service models and emergent ones. Each service is characterised by a different abstraction level and unit of scale.*

FaaS vs PaaS

Another common doubt regards the distinction between FaaS and PaaS models. Before FaaS, the application was the smallest unit of scale. PaaS providers would allow developers to scale their applications by deploying multiple instances, with at least one instance always available. The majority of PaaS vendors nowadays harnesses container technology to host users' applications more efficiently [80].

In contrast with PaaS, the FaaS model allows developers to break their application down into functions and scale each function independently. Differently from PaaS, functions may be scaled down to zero. For most programming languages, a function is the smallest unit of logic composing the application behaviour. Accordingly, functions are the ultimate back-end logic that can be abstracted by providers. Despite the similarities with the PaaS model, *FaaS lies one level beyond in terms of back-end logic abstraction*. Figure 2.4 highlights the differences between various cloud service models. In contrast with PaaS, functions (and not the application) is the customer managed unit of scale.

2.4.3 FaaS vs Microservices

The microservices architecture is a variant of the service-oriented architectural style. Applications that follow the microservices architecture are structured as a suite of loosely coupled, lightweight services, each running

its own process and communicating through lightweight mechanisms [54].

Among other benefits, the microservices modularity enables their development to happen in parallel using distinct programming languages, technologies, and tools. Moreover, microservices are deployed and scaled independently. The *tolerant reader* technique aims to reduce the likelihood of failures from unexpected inputs when the microservices interface evolves.

Also importantly, microservices are ideally built around fine-grained business capabilities by autonomous, multidisciplinary teams. This organisational approach contrasts with the more traditional one in which teams are organised around technology layers (e.g. a single database team responsible for multiple application modules).

The microservices architecture moves away from sophisticated solutions for message routing, choreography, transformation, and other functionality provided by tools such an *Enterprise Service Bus* in traditional service-oriented architecture. Instead, the microservices architecture favours *dumb pipes, smart endpoints*. HTTP and other standard web protocols are widely used. Each microservice parses a request, applies application logic and produces a response. Their choreography does not require complex protocols like BPEL. Alternatively, lightweight message bus like RabbitMQ enables reliable asynchronous communication.

The breakthrough in infrastructure automation is both driven and a driver of the microservices architecture. In one hand, the additional burden for developing, testing, deploying, and monitoring a large set of independent microservices largely depends on the use of automation techniques and tools. On the other hand, the adoption of this architectural style boosted the creation and evolution of the existing techniques and tools.

The *Continuous Delivery* software development discipline is commonly associated with the microservices architecture. This development approach makes use of intensive test automation that goes from basic unit tests up to integration, user acceptance and performance tests. The ultimate goal is to automate the deployment to the production environment in a continuous and safe manner.

Similarly to microservices (or even more significantly), functions are lightweight and fine grained and thus are likely to exist in large numbers. Although the management of infrastructure is abstracted away from developers, the latter still need to test their functions before releasing them to production. Hence, Continuous Delivery is also critical for the efficiency in which serverless functions are developed and evolved.

There is a clear synergy between FaaS and the microservices architecture. Indeed, both approaches are likely to co-exist as part of the same

application. Serverless functions could even be referred to as *microservices* [54, 58], as they are small and modular, communicate through lightweight protocols (often through a RESTful API), and are independently deployable by fully automated machinery (i.e., the serverless platform). Furthermore, the container technology is at the heart of both models. The intermediation performed by the serverless platform may, however, be seen as a form of *smart pipe*, which breaks an important principle of the microservices architecture. Given the particularities of the FaaS model, throughout this thesis we shall use the term *serverless function* or just *function*.

FaaS Platforms

Introduced in 2014, AWS Lambda [4] was the first FaaS platform to become widely adopted. It is also the first serverless offering by a major cloud vendor, followed by Google Cloud Functions, Microsoft Azure Functions and IBM's OpenWhisk [43] in 2016. Since then, several open source and commercial FaaS offerings appeared and continue to appear.

Among the existing open source FaaS platforms, Apache OpenWhisk [104] is considered as the most mature option. Originally developed as a commercial service by IBM, it was later on incubated by The Apache Software Foundation. IBM continues to back the platform development and uses OpenWhisk as the basis of its commercial Cloud Functions service. The project also receives an active contribution from the open source community. More recently, Adobe and Red Hat decided to adopt the OpenWhisk as their serverless platform. OpenWhisk architecture focuses on scalability and resiliency. It leverages cutting edge technology such as CouchDB, Kafka, Nginx, Redis and Zookeeper. The framework includes a REST API-based Command Line Interface along with another tooling to support packaging, catalogue services and many popular container deployment options.

Another popular open source platform is OpenFaaS. This platform dates from mid-2017, thus younger than OpenWhisk. Compared to OpenWhisk, OpenFaaS offers a lighter approach to FaaS. While OpenWhisk provides its implementation of a container orchestrator, OpenFaaS leverages state-of-the-art technology for this purpose, namely Kebernetes [4]. Moreover, OpenFaaS delegates the auto-scaling to external tools that monitor the workload and allow users to define auto-scaling rules in terms of the monitored metrics. Figure 2.5 shows the interest over time for OpenWhisk and OpenFaaS [5].

[4] https://kubernetes.io/
[5] https://trends.google.com

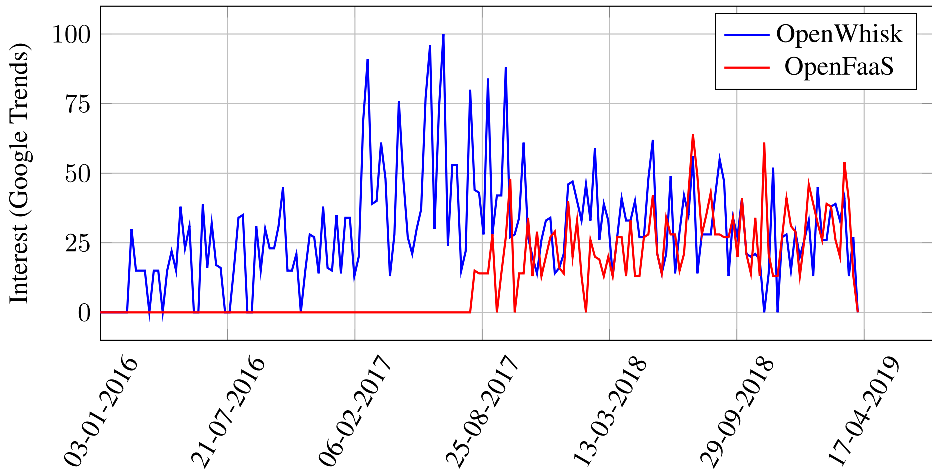


Figure 2.5: *OpenWhisk vs OpenFaaS — Interest over time measured by Google Trends*

Finally, OpenLambda is an academic effort towards an open source serverless architecture [38]. The platform consists of many subsystems that coordinate to run Lambda handlers, including a local execution engine that sandboxes handlers, a load balancer, and a distributed database. Although attractive from the research perspective, OpenLambda is still at an experimental stage and count on a few active developers.

2.5 Autonomic Computing

Autonomic Computing [47, 112] emerged as a paradigm to tame the growing scale, heterogeneity and dynamism (hence, complexity) of emerging computing systems. This paradigm drew its name and inspiration from the human autonomic nervous system. Its overarching goal is to realise computer and software systems that can manage themselves following high-level guidance from humans [81].

To be considered autonomic —similarly, self-managing or self-adaptive— a system should have the following properties [47, 91, 112]:

- **Self-configuration:** is a system's ability to readjust itself automatically and dynamically following high-level policies that specify what is desired but not how it is to be accomplished.

- **Self-healing:** is a system's ability to ensure software and hardware faults do not become failures (reactive mode), and to monitor its vital signs to prevent faults from occurring (proactive mode).

- **Self-optimisation:** is a system's awareness of its ideal performance combined with its ability to measure its actual performance and act to reduce or close the performance gap.

- **Self-protection:** is a system's ability to defend itself from accidental and malicious hazards, and to prevent localised failures to become system-wide.

The equilibrium of an autonomic system is impacted by both its internal (i.e., *self*) and external (i.e., *context*) environments [81, 91]. To achieve the properties above (or objectives [112]), the autonomic system must be aware of its internal state (self-awareness) and its external operating conditions (context-awareness). To this end, the autonomic system makes use of self-monitoring capabilities —through *sensors*— whereas self-adjusting is used to counter the effects of changes —through *actuators*. Together, these two activities form a *closed-loop control system* with the *feedback loop* aiming to adjust itself to changes during operation [91].

An autonomic system may comprise one or multiple autonomic elements [47]. Each autonomic element contains resources and delivers services to humans and other elements. Autonomic elements are typically composed of an *autonomic manager* (or *managing system*) and a *managed component* (or *managed system*).

The autonomic manager is responsible for a managed component within a self-contained autonomic element [112]. It monitors and detects changes in the internal state of the managed component and its external environment, and devise an adaptation plan based on the analyses of this information, taking into account the current goals and requirements of the autonomic element. The managed component consists of the functional, adjustable unit that performs the required services and functionalities.

The autonomic manager realizes the closed-loop control system. IBM represents this loop as the *Monitor*, *Analyse*, *Plan*, and *Execute* (MAPE) loop. The first part —monitor and analyse— process the data collected from sensors to achieve self-awareness and context-awareness. The last part —plan and execute— decides on the necessary self-management actions to be executed through the effectors.

The architecture of an autonomic element may vary [47, 112]. For instance, the autonomic manager and the managed component may be designed as a monolith. Alternatively, the autonomic manager may be provided externally, for instance, as an agent living in the same host as the managed component or in a different machine. In the latter case, interactions

occur remotely, often using a standard protocol (e.g. HTTP) and a consolidated distributed system architecture (e.g. peer-to-peer, client-server).

2.5.1 Adaptation Taxonomy

Autonomic systems are often classified according to a set of facets characterising their self-adaptation mechanisms, such as the origin of the change that triggers adaptation, the adaptation type and approach, the object to be adapted, and the adaptation time. Herein we summarise the key aspects found in the classification by Rohr et al. [89] and Salehie et al. [91].

Time

Time is an important aspect regarding the adaptation of an autonomic system. This facet is often refined as *reactive* and *proactive* adaptation [91]. In the reactive mode, self-adaptation occurs only after a change (e.g. in the expected system behaviour or performance) is perceived. In contrast, proactive self-adaptation anticipates changes, i.e., self-adaptive actions are taken beforehand based on change prediction [70].

Rohr et al. [89] further distinguishes *predictive* adaptation from *proactive* adaptation. While the former refers to the previously mentioned ability of the system to anticipate changes, proactive adaptation aims to keep one or more system metrics at better levels even if the current levels are acceptable. Hence, proactive adaptation is closely related to self-optimisation.

System Layer

The object of adaptation can reside at distinct layers of the autonomic system, namely: at the hardware, the operating system, the middleware, and the application layer.

First and foremost, a computing system comprises hardware resources such as CPU, memory, storage, and network components. Virtualisation technologies enable access to these resources to be controlled at different granularity levels. Physical and virtual resources may be added or subtracted from the system as a result of its self-management process.

An operating system usually manages the various resources composing the computing system and, depending on the case, a middleware platform. Each of these layers may also be subject to adaptation (often through reconfiguration) as part of the system self-management.

With the increasing complexity of software systems, the application layer has received significant attention from the autonomic computing and especially from the self-adaptive systems communities. Depending on the

architecture and technology used for its implementation, an application can be decomposed into services, methods, objects, components, aspects, and other sub-parts [91]. Each of these may be subject to changes as part of the system adaptations of different granularity levels.

Type

According to Salehie et al. [91], self-adaptation can be categorised into the following types: open/closed, model-based/free, and specific/generic adaptation.

Close-adaptive systems can perform a limited number of adaptive actions. In open adaptation, new behaviours can be introduced while the system is operating (i.e., at runtime).

In model-free adaptation, the autonomic manager adapts the system based on its requirements, goals, and alternatives and without a predefined model for the environment nor the system itself. Conversely, a model-based adaptation relies on models of the system and context to decide amongst adaptive actions. For instance, the system model may be based on existing theory (e.g. queue theory), architectural or domain-specific models.

Finally, a self-adaptation mechanism or method can be classified as specified or generic. In the former case, the self-adaptation targets a specific domain or application. In the latter case, the process may encompass multiple domains and applications.

Approach

Salehie et al. 's adaptation taxonomy also comprises an approach facet, which is refined into the following sub-facets: static/dynamic decision making, internal/external adaptation, and making/achieving.

A static-adaptation system has its decision process hard-coded. The modification of the decision procedures requires recompilation and redeployment of the affected components. Conversely, dynamic decision-making relies on policies, rules, or QoS definitions externally defined and modifiable at runtime.

Internal adaptation logic is intertwined with application logic. Problems of such approach include, but are not limited to, poor scalability and maintainability. It is useful for localised adaptations but less efficient when broader information regarding the internal state and context are needed. The external approach is in-line with the autonomic element architecture previously discussed. The autonomic manager (or managing system) implements and performs the adaptation process.

At last, Salehie et al. further classify the adaptation approach as making or achieving. These sub-facet refer to the strategy for introducing self-adaptivity into the system. According to Sterrit [112], the making approach follows software engineering methods. In contrast, the achieving approach implies adaptive learning and

Control Distribution

Depending on the autonomic element architecture —namely, monolithic or distributed— and the adaptation approach —namely, internal or external— the adaptation logic (or control [89]) can be categorised as centralised or decentralised.

In centralised self-adaptation, a single entity is responsible for all the autonomic operations in the system. This centralised control unit has a global view of the system, which facilitates the analysis and planning activities. On the other hand, the centralised approach creates a single point of failure. Moreover, it may result in substantial communication overhead and eventually become a bottleneck in large distributed systems.

Decentralised autonomic systems are characterised by the distribution of the adaptation logic over adaptation among its constituent autonomic elements. This approach is usually employed with large distributed systems. Each element makes decisions based on their local view of the system, which mitigates communication bottleneck. On the other hand, this limitation often reduces the efficiency of adaptation decisions.

A hybrid approach is also possible. Indeed, many systems use a combination of centralised and decentralised self-adaptation to compensate for the limitations of each approach. The adaptation logic is structured in a hierarchy of layers. The lower layers interact directly with the managed application to achieve a timely and effective adaptation, whereas the higher layers have a more comprehensive view of the system based on which more efficient adaptation decisions are made.

2.5.2 Multiple-Criteria Decision-Making

Multiple-Criteria Decision-Making (MCDM) is an important branch of decision-making theory [76, 86]. MCDM problems concern the selection of the best alternative(s) among many, or their ranking (sorting). All the alternatives are evaluated concerning multiple criteria chosen by the decision-maker(s).

According to Rezaei [86], a *MCDM problem* is generally shown as a

matrix *P*:

$$P = \begin{array}{c} \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{array} \begin{array}{c} \begin{array}{cccc} c_1 & c_2 & \cdots & c_n \end{array} \\ \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mn} \end{pmatrix} \end{array}$$

$\{a_1, a_2, \cdots, a_m\}$ is the set of available alternatives, $\{c_1, c_2, \cdots, c_n\}$ is the set of chosen decision making criteria, p_{ij} is the score of alternative a_i with respect to criterion c_j.

Since the goal of an MCDM problem is to select the best alternative(s) or to rank them, we need to assign to each alternative a_i an overall *score value* V_i. V_i can be obtained using various methods. In a general form, if we assign a weight w_j ($w_j \geq 0, \sum w_j = 1$) to each criterion c_j, then V_i can be obtained as follows:

$$V_i = \sum_{j=1}^{n} w_j p_{ij}$$

Usually the scores p_{ij} are objective and directly measurable (e.g. time, energy, monetary cost). Indeed, the difficulty of making decisions when multiple criteria are involved relies on determining a set of criteria weights $\{w_1, w_2, \cdots, w_n\}$ that accurately translate the decision maker(s) preferences.

CHAPTER 3

A Serverless Architecture for Multi-Access Edge Computing

3.1 Overview

As a starting point of the contributions in this thesis, we tackled the problem of defining an architecture for *Multi-Access Edge Computing* [16]. Among the various strands of edge computing, MEC has received significant attention from both industry and academia. Famous MEC use cases [103] include the offloading of computation and the pre-processing of data from consumer devices. MEC standardisation efforts have been carried by ETSI, who authored several documents, including a reference architecture [105]. Researchers from different areas have also tackled MEC from the architecture, communication, and offloading perspectives [61, 64, 83].

Like other edge-centric models, the decentralised nature of MEC imposes limitations regarding its capability of hosting all the applications and services that otherwise would be hosted by cloud data centres. An overloaded MEC server significantly degrades the user experience and negates the advantages of the edge computing paradigm [93]. Thus, MEC services might not be tackled with the straightforward migration of existing cloud models such as the user-centred deployment of virtual machines (IaaS) and

containers (CaaS).

Recently, *Serverless Computing* [31] appeared as a disruptive alternative that delegates the management of an application execution environment to the infrastructure provider. As a consequence, provider-managed containers are used to execute functions without pre-allocating any computing capability or dealing with scalability, servers configuration, and load-balancing burden [9, 38, 87]. By taking into account different aspects such as the need for an efficient allocation of resources, we propose a *Serverless MEC Architecture* that enables the offloading of computation from user devices to surrogate MEC nodes with *minimum latency* and *high throughput*.

The adoption of a serverless architecture should boost the utility of MEC nodes and the accessibility of MEC-based solutions, allowing one to deploy more functionality. Also importantly, the proposed architecture aims to cope with abrupt and unpredictable workload fluctuations resulting from the mobility and churn of devices. Furthermore, the Serverless MEC Architecture tackles some critical limitations in FaaS platforms with optimisations and supporting services.

The Serverless MEC Architecture described in this chapter is later on extended in Chapter 4 to include a broader range of deployment alternatives resulting from the combination of mobile, edge, and cloud computing. It is also the baseline for the framework targeting the self-management of densely distributed infrastructures and services presented in Chapter 5.

The remainder of this chapter is organised as follows. Section 3.2 describes some relevant requirements and provides further details about ETSI's Reference Architecture. Section 3.3 introduces the Serverless MEC Architecture, whereas Section 3.4 describes the instantiation of our architecture as a *Serverless MEC Platform*. Finally, Section 3.5 presents a recovery protocol for improving the resilience and availability of the proposed Serverless MEC Architecture.

3.2 MEC Architecture and Use Cases

This section sheds light on the requirements for services provided by MEC platforms. We refer to ETSI's MEC Use Case and Requirements [103] and additional literature. In particular, we summarise the most relevant use cases and requirements in the context of latency-sensitive applications, which are the primary target of the contributions presented in this chapter. This section also provides a more detailed description of ETSI's Reference Architecture for MEC [105]. In contrast with Section 2.3.4 —where ETSI's Reference Architecture was compared with other MEC architectures found

in literature— this section focuses on the functional elements and their interaction. Note that some of the terminology used by ETSI was adapted (e.g., *user device* instead of *user equipment* and *MEC node* instead of *MEC host*). This section concludes with the introduction of a Running Example.

3.2.1 ETSI Framework and Reference Architecture for MEC

The ETSI MEC framework is composed of three levels: the *mobile edge system level*, the *mobile egde node level*, and the *networks level*. The reference architecture contains both the functional elements and the reference points allowing interaction among them. In particular, ETSI's Reference Architecture focuses on the system and node levels (the network level is out of its scope). Next, we describe the core entities in ETSI's Reference Architecture for MEC relating to this work. The interested reader can find more detailed information in the original document [105].

Figure 3.1 depicts ETSI's Reference Architecture. Functional blocks may not necessarily represent physical nodes in the mobile network, but rather software entities running on top of a *Virtualisation Infrastructure*. The *Mobile Edge Orchestrator* is the core functionality in the MEC *System Level Management*. Among others, it is responsible for maintaining a larger view of the MEC system and selecting appropriate MEC nodes for application instantiation based on constraints (e.g. network latency).

As discussed later in this thesis, the proper orchestration of infrastructure and services is key for the scalability and performance of MEC and similar edge systems. Notwithstanding this, we opted for starting our contribution at the node level. Its main functional elements are:

- **MEC node:** an entity that contains a *Mobile Edge Platform* and a *Virtualization Infrastructure* providing compute, storage, and network resources needed for running *MEC applications*.
- **MEC Platform:** the collection of essential functionality required to run MEC Applications on virtualised resources; it also enables MEC applications to provide and consume *MEC services*.
- **MEC Applications:** instantiated on the *Virtualisation Infrastructure* (hosted by the MEC node) based on configuration and requests validated by the *MEC Platform Management*.
- **MEC Platform Manager:** responsible for the life-cycle of MEC Applications, for providing element management functions to the MEC Platform, and for managing application rules and requirements.

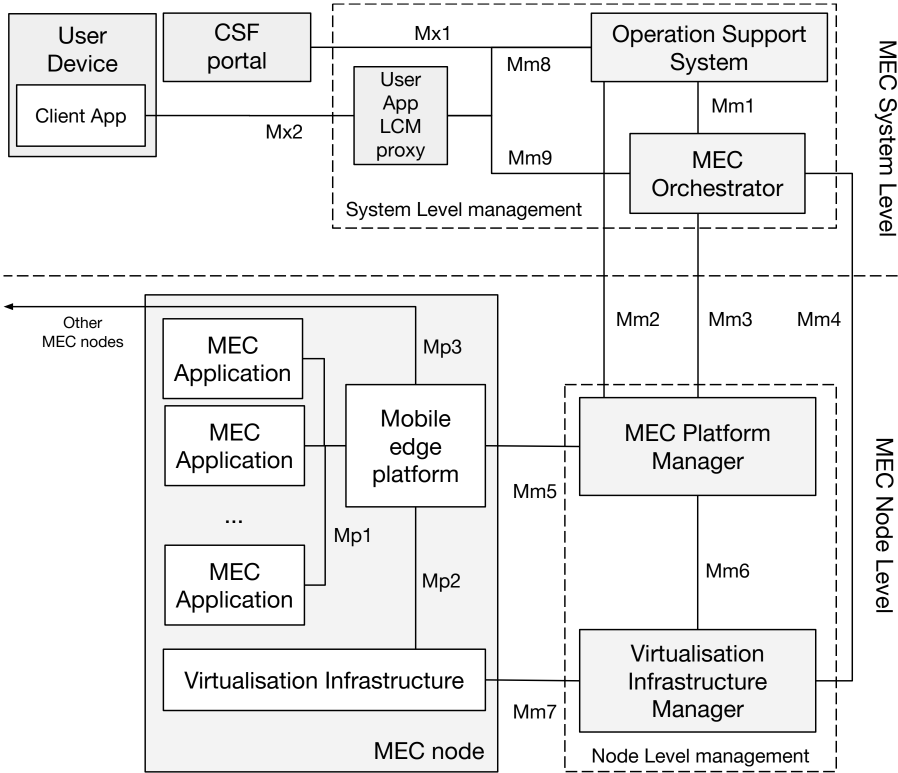


Figure 3.1: *MEC reference architecture [105]*

- **Virtualisation Infrastructure Manager:** responsible for allocating, managing and releasing virtualised resources from the *Virtualisation Infrastructure*; and for collecting performance metrics and detecting faults.

3.2.2 MEC Use Cases

Herein, we provide examples of use cases that are commonly associated with MEC by ETSI [103] and researchers from various fields [61, 64, 83].

Augmented Reality

Augmented Reality may be defined as the combination of the real world view and supplementary computer-generated information [41]. This technique allows users to have additional information from their environment by performing an analysis of their surroundings, deriving the semantics of the scene, augmenting it with additional knowledge, and feeding it back to

the user within a short time. The device can be for example a smartphone or any wearable device with a camera and other sensors (e.g. compass).

Assisted Reality

Assisted Reality uses a similar method, but its purpose is to actively inform the user of a matter of interest to him (e.g. danger warning, conversations). This information might be used for example to support people with disabilities (e.g. blind, deaf, of old age) to improve their interactions with their surroundings.

Virtual Reality

Another correlated use case is Virtual Reality. Its purpose is to render the entire field of view with a virtual environment either generated or based on recorded/transmitted environments. This information might be used for example to support gaming implementations or remote viewing while using the most natural input device available.

Cognitive Assistance

Cognitive Assistance takes the concept of Augmented Reality one step further by providing personalised feedback on activities the user might be performing (e.g. recreational activities, furniture assembling, cooking). Also in this case, the analysis of the scene and the advice to the user are sensitive to delay.

Image and Video Editing

The emergence of mobile devices equipped with a powerful camera combined with the massive adoption of social network applications boosted the number of pictures and videos produced by amateur and professional end-users. Such popularisation has been accompanied by new mobile applications that allow users to customise images and videos to their taste. These applications often rely on computationally-intensive procedures based on image processing and, more recently, neural network techniques.

3.2.3 MEC Requirements

ETSI specifies a set of requirements for the MEC framework and architecture [103, 105]. Herein we describe the most important ones in the context of the main contribution in this chapter, which falls in the category of a

MEC Service. Other requirements, including some concerning *MEC Applications*, are also presented since they are closely related with the *Computing Services* we propose.

- **Life-Cycle:** A third-party may request the deployment of an application to the MEC system.
- **Life-Cycle:** Applications must be seamlessly deployed on different MEC nodes without specific adaptation from the application.
- **Life-Cycle:** Upon request, a client application needs to be provided with a service instance.
- **Life-Cycle:** The client application may specify requirements such as *latency* and *compute resources* that must be fulfilled by the MEC node.
- **Life-Cycle:** A new instance must be created if it is not yet running.
- **Life-Cycle:** An instance must be terminated if it is not used.
- **Connectivity:** the MEC platform will allow authorised MEC applications on the same host to communicate with each other and with external third-party services.
- **Mobility:** The MEC system will be able to maintain connectivity when the user device performs a handover to another cell (i.e., another base station) assisted by the same or a different MEC node.
- **Mobility:** The MEC system needs to support the continuity of the service when the user device performs a handover to another cell assisted by the same or a different MEC node.
- **Architecture:** Application components may reside outside of the distributed edge cloud, for example in the device or central cloud.
- **Architecture:** Different deployment scenarios must be supported, including base stations, aggregation points, and infrastructures at the edge of the Core Network (e.g. distributed data centres and ISP Gateways).

3.2.4 Running Example

In this section, we introduce two application scenarios that are used as a running example throughout the remainder of this chapter.

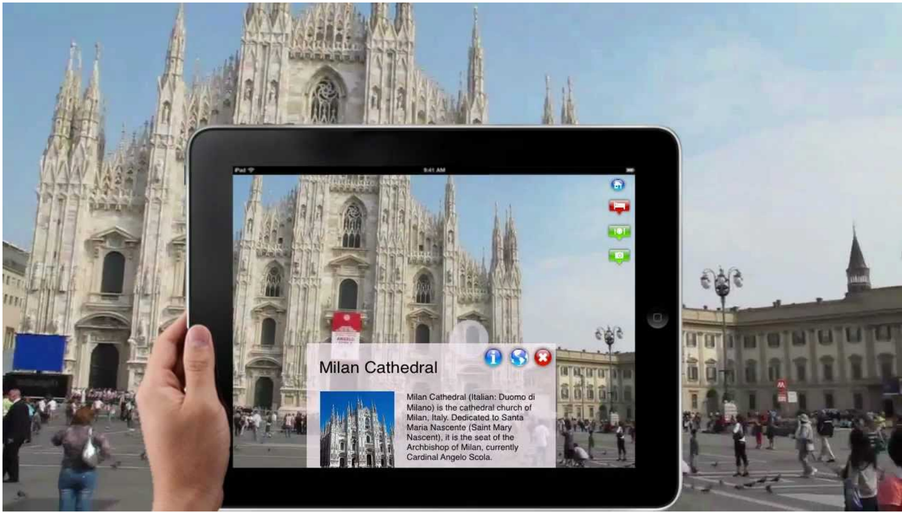


Figure 3.2: *An example of a Mobile Augmented Reality app* [1]

.

Mobile Augmented Reality

Mobile Augmented Reality (MAR) [50, 118] emerged as the fusion of Augmented Reality and mobile computing. MAR is an example of applications for which low latency and high throughput are key requirements. MAR applications enrich the interaction of users with the physical world by:

- augmenting their vision of the reality with relevant information (e.g. historical information about buildings and monuments);
- modifying it (e.g. with translations of the captured text in a different language);
- adding virtual elements that can mimic interactions with the real world (e.g. virtual objects or creatures from a fantasy game); or
- helping users fulfil physical tasks (e.g. with the highlight of free parking spots).

Our example MAR application is conceived to help the tourists that visit a city and want to receive relevant information about points-of-interest (POI), such as monuments, buildings, and other architectural elements, by looking at them through their mobile devices (see Figure 3.2) or special glasses. Based on approaches found in literature [42, 84, 118], the following steps summarise the sequence of tasks in MAR applications:

[1] Accessed from `http://etips.com`

1. The reality that must be augmented should be captured by using the device's camera.

2. The captured scene must be scanned and processed so that *features* from POI in the scene can be detected and extracted.

3. Additionally, the scene may be contextualised by tracking the device's orientation and geographical location [84].

4. POI features are then matched against those in a feature dataset [118] or a neural networks trained model [84].

5. Information associated with the identified POI must be retrieved, often from remote servers (due to its size and dynamic nature).

6. Finally, the application augments the real world view with computer-generated content, which is displayed on the device's screen.

As users can rapidly move and target different portions of the world around them, frames from the target scenes must be captured by the device's camera at a fast rate (step 1), generating a significant volume of data. The extraction of features from the objects in these frames (step 2) relies on compute-intensive image processing [25]. Undesirable or prohibitive network delay (and traffic) can be avoided by letting steps 1 to 4 and 6 be performed locally and only delegating step 5 to services in the cloud [42]. However, this kind of approach may fail to meet users' expectations because it can significantly reduce the battery level of their devices [20, 25]. Also importantly, continuously fetching POI information from cloud services could be slow and disruptive for the user experience.

The offloading of computation from mobile and other IoT devices to MEC platforms rather than using standard cloud services should bring several advantages. First, it provides the low latency and high throughput required by real-time and interactive applications. Second, it prevents overloading resource-constrained devices with compute-intensive tasks. Third, it prevents congesting the network with large volumes of data to more remote cloud data centres. Last but not least, relying on densely distributed MEC nodes instead of centralised cloud services is advantageous since supplementary information concerning POI (and other types of city assets) is highly localised; as showed elsewhere [25], a reduced and geo-located dataset can significantly improve the responsiveness and accuracy of the object detection technique employed by many outdoors MAR applications.

Figure 3.3: *Facial Recognition used for detecting suspects in a crowd*[2]

Facial Recognition

Amongst other use cases, facial recognition [68,98] is used as a non-invasive and scalable *biometrics* technique that aims to provide the identity of one or more individuals depicted in an image, be it a photograph or a video frame from a surveillance camera.

The use of computers in the recognition of subjects dates back to the 1960s using a man-machine approach in which facial landmarks were manually annotated and then automatically computed and compared between images to determine identity. Since them, multiple techniques were developed; the accuracy of state-of-the-art facial recognition methods is orders of magnitude higher compared to the pioneer attempts. From a more abstract level, real-time facial recognition techniques can be summarised by the following steps [82]:

1. A digital image or video frame containing one or multiple individuals is captured.
2. All the faces in the image are detected and located.
3. Each sub-image containing a detected face is pre-processed to obtain a canonical alignment and orientation of the subject's face and thereby improve the performance of the subsequent recognition procedures.
4. Facial features are extracted and selected.
5. Facial features are then compared against a broad set of features from wanted individuals (e.g. a trained set [98]).

[2] Accessed from `www.digitaltrends.com/cool-tech/facial-recognition-china-50000`

6. The identity of each person is output along with the confidence value.

A Facial Recognition application designed to identify wanted individuals out of crowd would also benefit from the MEC architecture. In contrast with the MAR application, the central advantage is not to reduce battery consumption nor minimising the delay, but to prevent that large volumes of data to be transported all the way to cloud data centres [18]. To this end, MEC nodes lay in the middle of a three-level architecture; data-intensive tasks such as face detection (step 2), pre-processing (step 3), and feature extraction (step 4) are delegated to the MEC platform. In turn, the feature matching procedure (step 5), which relies on a database containing a broad set of features, is performed by a cloud service.

3.3 The Serverless MEC Architecture

In this section, we introduce the building blocks of a *Serverless MEC Architecture*. We start with the introduction of *Self-Managed Computing Services*, which enable the offloading of application logic from mobile and IoT devices. In sequence, we present the supporting services and optimisations needed to overcome performance limitations in FaaS platforms and to satisfy the requirements from latency-sensitive applications.

3.3.1 Self-Managed Computing Services

While MEC servers are ideal candidates for offloading compute-intensive tasks needed to preserve device resources and kill latency, these servers are themselves potentially constrained [61]. Accordingly, the feasibility of hosting dedicated VMs (IaaS) or even containers (CaaS) would be limited.

To overcome this limitation, we propose to extend the MEC Platform in ETSI's Reference Architecture with the components from a serverless architecture. The resulting *Serverless MEC Architecture* provides support for the deployment and execution of functions (FaaS). We refer to this kind of service as *Self-Managed Computing Services* or just *Computing Services*.

The Computing Services proposed are fundamentally different from the concept of MEC Applications in ETSI's framework. Although the initialisation and termination of MEC Applications on a request basis is part of its specification, ETSI depicts MEC Applications as long-living instances outlasting requests from one or multiple clients. Also, ETSI makes explicit reference to the provisioning of virtual machines and containers to host MEC applications, thus similar to a typical cloud-based IaaS/CaaS deployment. In contrast, we propose MEC Computing Services to follow a

serverless architecture through the *Function-as-a-Service* model [9,38,87].

Serverless functions are stateless, lightweight by design and potentially ephemeral (may last for a single invocation). Moreover, the life-cycle management of functions and the orchestration of containerised compute runtimes is abstracted away from developers and delegated to the *Serverless MEC Platform*. As a result, the use of computational resources is optimised, which allows more functionality to be deployed and more requests to be processed concurrently. Also importantly, the Serverless MEC Architecture exempts application providers and developers from manually performing infrastructure management tasks such as server and network configuration. According to Satyanarayanan and his colleagues [97], self-management is vital for the widespread deployment of cloudlets infrastructures. We argue that the same principle holds for MEC deployments.

In our Running Example, the MAR application relies on two compute-intensive tasks: one for *feature extraction* and another for *feature matching*. The first handles the extraction of features from captured scenes, while the second matches these features against a dataset [118] or a trained model [84]. An additional *POI information* task fetches computer-generated elements to be added to the scene rendered by the device's display. The Serverless MEC Architecture exploits the FaaS model to allow tasks such as *feature extraction*, *feature matching* and *POI information* to be written as serverless functions and consumed as web services.

Serverless functions could be referred to as *microservices* [54, 58], as they are small and modular, communicate through lightweight protocols (often through a RESTful API), and are independently deployable by fully automated machinery (i.e., the serverless platform). Nonetheless, we opted for keeping the consistency with the more commonly used terminology among serverless computing and FaaS communities.

The resulting application architecture comprises two fine-grained and highly cohesive functions: one for detecting POI in the captured scene, and another for retrieving POI information (e.g., from a *MEC Database Service*). Alternatively, the MAR functions could comprise a single, coarser AR function; its main advantage would be the reduced communication overhead. However, in the context of decentralised infrastructure, separating functions into distinct services has the following benefits: (i) it allows functions to be *deployed* and *scaled* independently (e.g. by heterogeneous MEC nodes); (ii) it allows functions to be *consumed* independently (e.g. by client applications hosted by different devices and by distinct applications).

The advantages above are particularly important since MEC nodes in different deployment configurations diverge in their computational capa-

bilities [61, 77]. The decision of where to execute each function (i.e., a MEC node in proximity or more powerful, but distant nodes) will depend on factors like the requirements of each function, the computational capabilities and resources available, as well as the network latency and throughput among devices and MEC nodes hosting the Serverless MEC Platform.

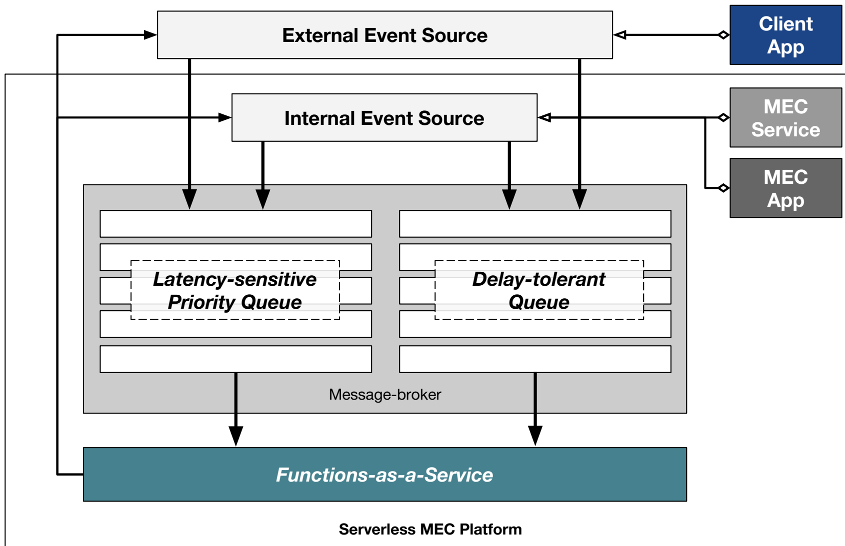


Figure 3.4: *Serverless MEC Architecture. Events from external and internal sources are fed into a message-broker component. Latency-sensitive events are processed with higher priority, whereas data-intensive, delay-tolerant events may stay longer in queue.*

Serverless architectures are, by design, event-driven. A typical use case among cloud vendors is to enable functions to be executed in response to events such as the upload of an image to a storage service. Indeed, functions can become consumers of various sort of events generated by an ecosystem of back-end services and applications, including other functions.

In the context of MEC, an event-driven architecture is key for the integration of different MEC applications and services. Not only functions are activated by an external source of events —namely, requests from latency-sensitive applications— but also in response to events that are internal to the Serverless MEC Platform. Besides the trivial example in which functions trigger one another, a message-broker component provides loosely coupled integration between functions and other MEC subsystems and services. Our Serverless MEC Architecture harness the *priority queue* functionality provided by this component to enable the in-transit processing and analysis of data from data-intensive, delay-tolerant applications.

Back to our Running Example, the Facial Recognition application relies on Serverless MEC Platforms to perform a preliminary analysis and identification of facial features from wanted individuals among the crowd. More specifically, the Serverless MEC Platform provides the compute runtime for the execution of *face detection*, *face alignment*, and *face recognition* tasks, which are data-intensive, but not sensitive to sub-second delays. Contrary to the MAR application, upon preliminary facial identification, the results are sent to the cloud instead of returning to the external event source.

Last but not least, the MEC architecture provides the critical advantage of data locality. Such advantage has three aspects: (i) fewer data must be cached on each MEC node; (ii) the latency of fetching updated data (e.g. from a cloud service) is substantially reduced [1]; and (iii) a reduced dataset often improves the performance of operations over these data.

For the MAR application in our Running Example, data locality restricts the scope of the POI identification function; the platform will need to fetch and store data only regarding the POI within the region served by its MEC node (e.g. accessed by client applications through the co-located base station [61]) instead of the broader area typically covered by cloud-based services. Also importantly, a reduced feature dataset improves both the performance and robustness of the feature matching procedure [25].

3.3.2 Supporting Services and Optimisations

The adoption of a serverless architecture offers many benefits in terms of granularity, delegation of infrastructure management, and resource allocation efficiency. Notwithstanding this, the satisfaction of more strict requirements from latency-sensitive applications by a Serverless MEC Platform demands further optimisations. Next, we propose additional services and mechanisms targeting the mitigation of overhead.

Container Initialisation Overhead

As introduced in Section 2.4, FaaS platforms implement different optimisation strategies to minimise the time needed to scale-out containers (cold start). Typical strategies include the caching of function assets, prewarming of uninitialised containers, and retention of idle (warm) containers for subsequent invocations.

Contrary to cloud deployments, the proximity with end-users in MEC architecture favours client awareness. For instance, ETSI specifies a *Location Service* that provides location-related information regarding user de-

vices currently associated with the MEC node. A similar kind of service might allow the Serverless MEC Platform to become aware of which applications are currently hosted and executed by user devices within its service area. The fact that client applications are already supposed to identify themselves to the MEC system corroborates this hypothesis.

The Serverless MEC Platform harnesses the client awareness discussed above to anticipate the *warming* of virtualised infrastructure and thereby mitigate the impact of cold start in latency-sensitive applications. More precisely, we propose the following policies:

- By default, all deployed functions are scaled from zero.
- Whenever resources are available, the presence of devices hosting active client applications that depend on functions provided by the platform will trigger the allocation of their respective compute runtime.
- The allocation strategy may be more conservative (e.g. one warm container per function) or aggressive (e.g. proportional to the number of distinct clients in the MEC node area).

MEC operators may also choose to enact different policies according to the SLA category of each function. For instance, functions with more strict requirements for latency may be associated with specific SLA that assure more aggressive pre-warming behaviour (provided with a higher cost).

Function Initialisation Overhead

In addition to cold start, the overhead in function initialisation must also be considered. For instance, the majority of FaaS platforms available — including Amazon's AWS Lambda, IBM's Cloud Functions, Google's Cloud Functions, and Microsoft's Azure Functions— enforce a programming model in which:

- Functions are entirely stateless, i.e., they have no access to internal state, and its output depends solely on its input and implementation.
- Function sources and dependencies are limited in size.
- Functions have access to a temporary folder within its containerised execution environment that can be exploited for caching data.
- Upon activation, a function sets up whatever dependencies it may have (e.g. a trained model used for object detection in MAR).

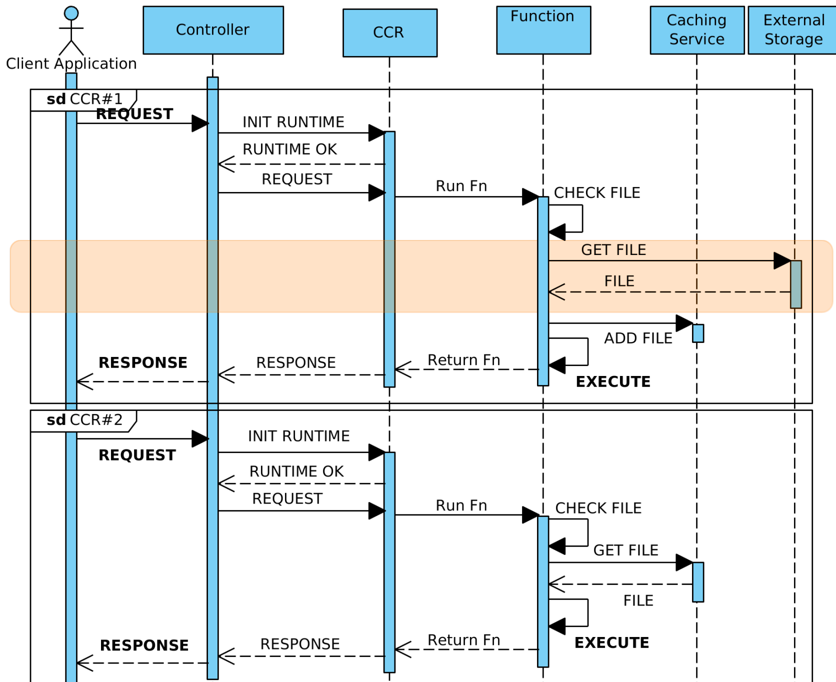


Figure 3.5: *Upon a first function activation (in CCR n° 1), the POI dataset is fetched from a cloud-based storage service and added to the CCR's temporary folder; subsequent first activations in other CCRs are served by the platform's Local Caching Service.*

- Subsequent executions can happen within the same or different containers, including fresh (cold) containers.

The primary motivation for limiting package size is to keep at a minimum the time taken to load the function's assets into the container environment and make it available for execution, which in turn enables functions to be allocated and scaled in a timely and efficient manner. Consequently, functions that depend on more massive datasets need to retrieve them from external sources when they are first executed within a fresh container. In this approach, the container's temporary folder is often used to mitigate networking overhead for subsequent executions.

Caching Service

In the context of latency-sensitive applications, retrieving extensive data sets may add prohibitive overhead. For instance, in the MAR example, a catalogue (circa $100MB$ [84]) is used for matching POI in the captured scene. To address this concern, Serverless MEC Platforms are equipped

with a *Caching Service*. This service allows functions to fetch data from external storage and keep them available at the MEC node for mitigating networking overhead.

The diagram in Figure 3.5 depicts the interplay between two fresh containers (CCR#1 and CCR#2) (initialised with the POI identification function) and the platform's Caching Service. Upon a first initialisation (CCR#1), the function fetches a POI catalogue from some remote cloud storage and then adds it to the local platform's cache. Subsequent container initialisations (e.g. CCR#2) retrieve the cached file from the Caching Service.

In-Memory Caching

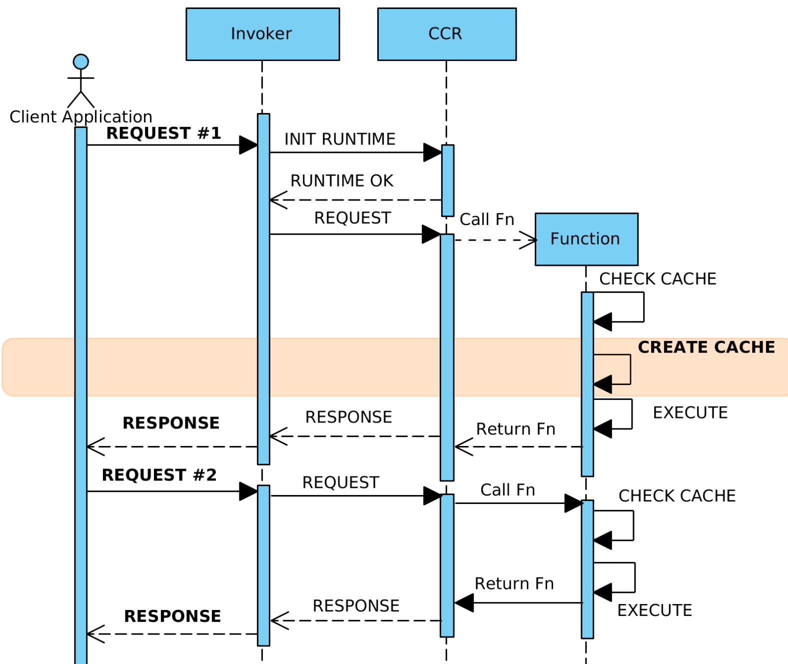


Figure 3.6: *In-memory Cache Mechanism. At the first request, the function assigns to the cache (a global variable) some complex object. Subsequent executions bypass time consuming creation of complex objects by retrieving them from the in-memory cache.*

While the aforementioned approach is adequate for many use cases, it may not be sufficient for latency-sensitive applications. For instance, the time needed to load complex objects into the program's memory before execution may be significant or even disruptive. Targeting this scenario, we propose a succinct, but effective modification to the FaaS programming model, namely the support for *in-memory cache*.

The rationale behind the proposed modification is directly related to the allocation strategy adopted by most FaaS platforms, i.e., the retention of warm containers for subsequent activations. While users are charged proportionally to the (memory) resources allocated only for the function execution duration, idle containers *will also retain memory*. Hence, application developers may opt to use part of the function's *memory quota* (a specified requirement) to keep *in-memory cache* across activations.

Figure 3.6 illustrates the in-memory cache mechanism within the function life-cycle. At each execution, the function checks the *cache* value. The first function execution populates the cache using the object's constructor. Subsequent activations of the same function in the same container will find the cached object initialised and ready for use. As a result, the performance of subsequent executions is much improved.

Going back to our running example, the MAR function exploits the in-memory cache mechanism to eliminate the overhead of creating and loading the heavyweight feature dataset used for POI identification at every function invocation, which in turn reduces the function response time.

It is important to note that the proposed mechanism does not conflict with the concept of function statelessness since the function output will solely depend on its input and implementation. The cached data will be shared among subsequent function activations (possibly from different clients). The FaaS platform can enforce its initialisation to be final to prevent misuse (e.g. using the in-memory cache to hold session data). Conversely, distinct functions will not be able to access each other's cache as they are never executed by the same container.

Function Workflows

The adoption of caching strategies is critical for the mitigation of function set up overhead. However, single function invocation may add significant network overhead whenever two or more functions form an execution flow. To avoid it, developers would be forced to either chain function calls through hardcoded dependencies, or write coarser and less cohesive functions. In the former case, function chaining also imposes a cost overhead, since caller functions are also charged for the time they are kept waiting.

To address latency, design, cost, and deployment concerns, the Serverless MEC Platform supports *function workflows*. The latter allows developers to define, through visual or textual programming, a function execution flow. Similarly to single functions, workflows are activated by events (e.g., an HTTP request). At each intermediary step, the result from the current function is passed to the next; the client application receives a response

solely from the last executed function, preventing the round-trip latency overhead resulting from multiple invocations.

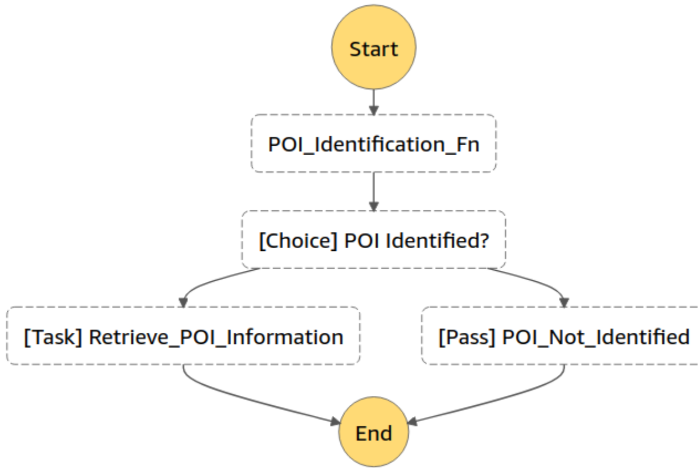


Figure 3.7: *MAR function workflow. The workflow starts with the POI_Identification_Fn, which process the scene frame and tries to identify POI in it. Upon identification (Choice state), a second task (Retrieve_POI_Information_Fn) retrieves information concerning the identified POI; otherwise it bypasses this task (Pass state) and returns.*

Note that FaaS is event-driven by design. One may also opt to coordinate the execution of functions by mapping specific events —fired by one or more functions— with the trigger(s) of other function(s). Nonetheless, a workflow language provides a more straightforward solution for the modelling of execution flows with clear temporal order, causality and cardinality. Moreover, while event-driven coordination is asynchronous and results must be fetched asynchronously, the workflow orchestrator may perform blocking invocations to the platform's interface and pass along execution results to the next function without further delay.

Figure 3.7 illustrates a workflow for the MAR application. In this representation, we adopted the semantics defined by the *Amazon States Language*[3], which allows the specification of workflows as state machines. This language is also used by *AWS Step Functions*[4], a graphical workflow modelling language by the same vendor.

The first function (*[Task] Retrieve_POI_Information*) is responsible for processing the video frame sent by the device and identifying eventual POI, as described in Section 3.2.4. More specifically, this function outputs a map containing the object (i.e. a POI) along with its coordinates in the frame.

[3]https://states-language.net/spec.html
[4]https://aws.amazon.com/step-functions/

Upon identification (*[Choice] POI_Identified?*), the workflow proceeds by activating the second function (*[Task] Retrieve_POI_Information*). The latter receives as input the map produced by the first function; for each identified POI, it will retrieve computer-generated content to be superimposed to the original scene. Note that the workflow will return earlier (*[Pass] POI_Not_Identified*) if the first function has identified no POI.

State at the Edge

Thus far we have dedicated attention to stateless functions provided by Serverless MEC Platforms. The platform's Self-Managed Computing Services enable mobile and other IoT devices to offload latency-sensitive and data-intensive tasks. While the offloading of intensive computation is one of the central use cases, equally important is the mitigation of delay in accessing data. This is especially true for applications with *data locality*.

Take as an example the MAR application. Most of the information pertaining city POI are highly localised. The *POI_Information* function would greatly benefit from accessing these data locally, i.e., from a Database Service composing the Serverless MEC Platform's functionality. Addressing this kind of scenario, the Serverless MEC Architecture may comprise a *Database Service*. In contrast with the Caching Service, the Database Service provides the functionality of a SQL or NoSQL database.

Similarly to the Caching Service, the Database Service may enforce access to be read-only. Moreover, each platform stores a local view of geo-localised data (e.g. only POI in that area). Whenever a client application issues a *Command* (CMD) to update state, the latter is propagated to the cloud persistence layer, which holds a global view of the system. Conversely, client *Queries* (QRY) are promptly replied.

While POI information is less dynamic, the same is not true for other use cases that require access to volatile data with minimum delay. For example, let us think of a Mobile Multiplayer Game (MMG). Following a conventional multi-player architecture, an authoritative service is designed to host the game state and logic. This solution has two advantages. First, it mitigates $N - to - N$ communication overhead among peers at each user interaction. Secondly, it allows a reliable server to manage changes to the game state. The latter task may not be safely delegated to clients, who otherwise could modify the local game logic and state to their advantage. As an interactive real-time application, low-latency is a first-class requirement that justifies the deployment of this stateful service to MEC nodes.

The MMG application poses additional challenges. Contrary to the Caching and Database Services, game state is updated bottom-up follow-

ing client commands. Game session state tends to be transient and therefore easier to manage. Nonetheless, users may be scattered around distinct base stations and MEC nodes in the same area.

Two alternative solutions could be employed with the previous scenario: to create a single instance per game session (hosted by a single MEC node) or to spawn multiple instances. The former approach implies the choice of which node to host the session. Due to mobility, it might need to be migrated at runtime to a best candidate node (e.g. lower delay). The latter approach implies the synchronisation of state among different MEC nodes.

The replication of state across geographically distributed components is not trivial. The CAP theorem [33] states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees: *consistency*, *availability*, and *partition tolerance*. For example, if the delay between two MEC nodes at different localities is not negligible and data must be promptly available when requested, it is not possible to guarantee the consistency of data at each node.

In light of this, the single-instance approach is favoured. Indeed, application instance migration is part of ETSI's Use Cases for MEC [103]. It is also addressed by many authors who tackle the problem of placement and migration of services in edge-centric architectures [61, 125].

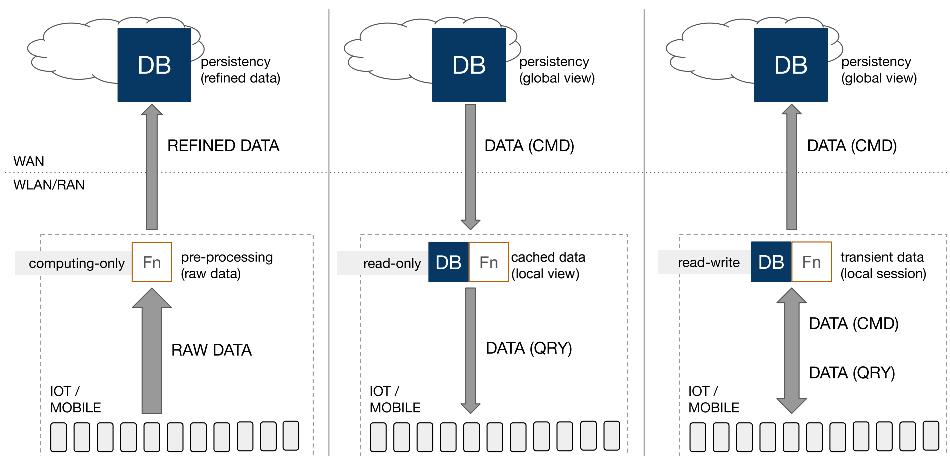


Figure 3.8: *Application scenarios involving data access/manipulation. Leftmost: computation-only targets data pre-processing. Middle: data-only substantially mitigates data access (QRY) overhead through caching. Rightmost: read-write enables transient data to be promptly modified and accessed (CMD+QRY) through sessions.*

Figure 3.8 illustrates the three scenarios involving data at the edge. The first scenario represents stateless computation in which raw data is pre-

processed at the edge and persisted by cloud data centres. The second scenario adds a read-only Database Service for mitigating the overhead of accessing data that may be cached at the edge. The third scenario enables transient data to be promptly modified and accessed by clients within a session from a latency-sensitive application (e.g. a real-time battle session from a MMG application).

3.4 The Serverless MEC Platform

In this section, we describe the instantiation of our Serverless MEC Architecture using ETSI's Reference Architecture. In particular, we focus on the functional elements comprising the Serverless MEC Platform and their interplay with native components and services in the ETSI specification.

3.4.1 Platform Architecture

The Serverless MEC Platform materialises the *MEC Platform* component in ETSI's Reference Architecture. The platform herein described aims to be comprehensive regarding the services and optimisations introduced in the previous sections. It also leverages the functionality to be provided by implementations of the MEC node and system, as specified by ETSI. This is particularly so for the features that are out of the scope of this work, e.g. security and authentication of users, service registry, and management of traffic rules. Whenever appropriate, we shall comment on how the Serverless MEC Platform integrates with other MEC components and services.

While academic efforts to the realisation of a serverless platform exist [39], these are yet far from the maturity level achieved by existing open source solutions. In light of this, we adopted OpenWhisk [104], a leading open source FaaS platform, as the serverless architecture of reference.

Triggers and Events

Figure 3.9 presents the Serverless MEC Platform architecture. Its entry point are the *triggers* associated with *events* by means of *rules*. As discussed in Section 3.3.1, events may be of different kinds and may come from external (e.g. HTTP requests) and internal sources (e.g. generated by other functions or platform services).

While event-driven architectures are typically asynchronous, real-time and interactive applications need results to be available as soon as possible. The Serverless MEC Platform tackles this requirement by supporting synchronous (blocking) invocations to deployed functions.

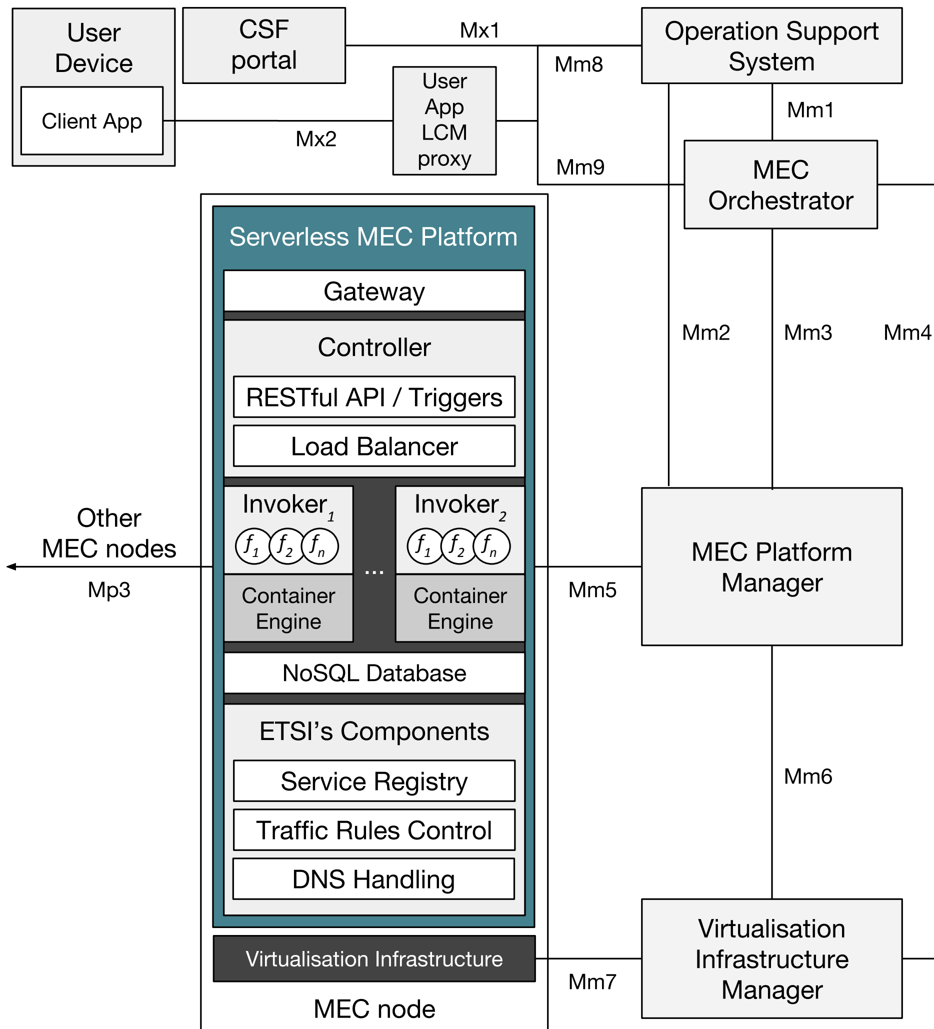


Figure 3.9: *Detailed view of the Serverless MEC Architecture integrated with the components from ETSI's MEC Reference Architecture [105].*

In the MAR application, an event that activates the POI Identification function consists of the upload of a *video frame* from the scene captured by the device's camera. Since it consists of an external invocation, the activation is triggered by the arrival of an *HTTP Request* to the *RESTfull API* exposed by the platform's *Controller*. Moreover, the request consists of a synchronous invocation; as soon as the execution finishes, results are sent back to the client application (as an *HTTP Response*).

For many types of applications, asynchronous invocations can be advantageous. Indeed, this is a common requirement for the in-transit data processing and analysis use case. For example, requests from the smart cameras in our Facial Recognition are one-way only, i.e., they do not expect analysis results to return.

The Serverless MEC Platform supports this modality through the *NoSQL Storage* component (a highly available non-relational database), which enables results to be fetched asynchronously through the unique identifier returned as response for each triggered execution.

Asynchronous invocations also enable execution results to be consumed a posteriori. Furthermore, it allows for the debugging and auditing of the function execution. For example, among the various MEC services that may comprise the Serverless MEC Platform, one is to allow for third-party administrators to request, via *CSF Portal* in ETSI's Reference Architecture, reports about the real-time or historical usage of the MAR application on various regions.

The platforms deployed to distinct MEC nodes push their activation log to the (global) MEC system database at two circumstances: (i) upon resource contention (e.g. MEC node storage is low); (ii) when network traffic is lower (e.g. early in the morning). The access to the function execution log follows a caching approach: upon a *cache miss*, the function execution log is retrieved from the corresponding Serverless MEC Platform (thus, in the opposite direction of the caching approach in Section 3.3.2).

Function Activation and Load Balancing

Once a request reaches the platform, it is forwarded to the *Controller* component. The latter identifies the function being called and triggers its activation. Note that, at this point, the request has been authenticated and admitted by MEC's *User App LCM Proxy* and forwarded to the Controller's IP, accordingly to the platform deployment set-up within the MEC infrastructure and the DNS rules defined by the *MEC Platform Manager* [105].

Upon activation, the Controller will proceed by delegating the function execution to an *Invoker*. Its goal is to manage the life-cycle of the containers

in which functions are executed (see Section 3.4.1). Depending on the MEC node's capabilities, one or multiple *Invokers* may be deployed to virtual machines provided by the *Virtualisation Infrastructure*.

Whenever multiple Invokers are deployed (see Figure 3.10), the Controller's *Load Balancer* module is responsible for selecting, upon each activation, an Invoker for handling the function execution. The Controller also keeps track of the Invoker health through periodic *heartbeats* and new Invokers may be deployed on-the-fly without disrupting platform behaviour.

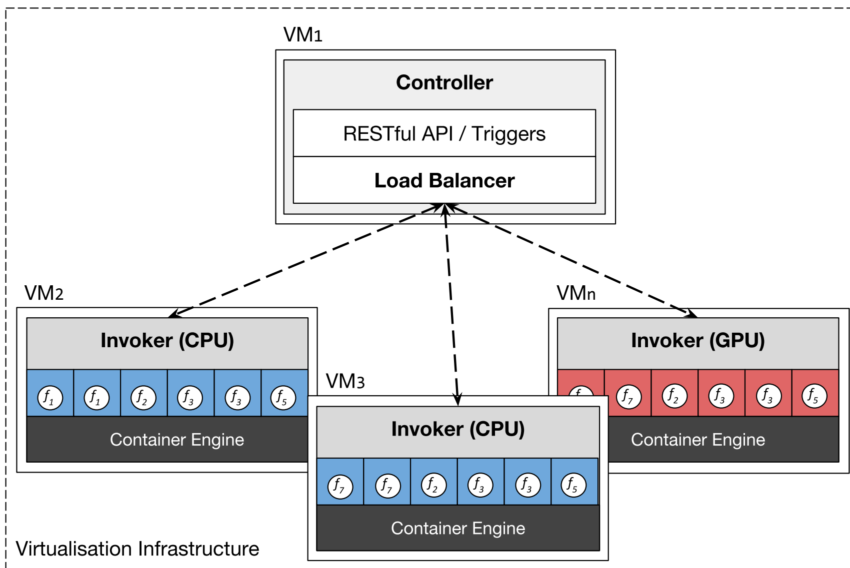


Figure 3.10: *Multiple invokers hosted by heterogeneous VMs provided by the Virtualisation Infrastructure comprising a single MEC node.*

A possible approach for load balancing would be to evenly distribute the workload among Invokers, e.g. using a round-robin algorithm. However, consolidating function activation has the advantage of increasing the rate in which warm containers are re-used [58], which is critical for resource allocation efficiency. This can be solved by mapping functions to a priority list of Invokers (e.g. randomised with a hash function to prevent collisions). A different Invoker may always be selected (with the increasing probability of incurring in cold start) in case the current top priority Invoker becomes overloaded or unavailable. Accordingly, the Controller's Load Balancer uses the following scheduling algorithm for each deployed function:

1. Upon a first activation, the controller maps the function to a randomised priority list using a hash function to prevent collisions.

2. The Virtualisation Infrastructure Manager monitors the CPU and memory usage level of the virtual machine(s) hosting platform's Invoker(s).

3. The Load Balancer schedules workload according to the priority list and the Invokers' status; *overloaded* and *unavailable* Invokers are skipped following the priority order.

4. In case all Invokers are overloaded, the workload is scheduled to the top priority Invoker, which will queue incoming activations.

Resource Management and Function Execution

The platform's Invoker allocates containers on an event-basis and according to the resources made available by the *Virtualised Infrastructure Manager* on behalf of the Serverless MEC Platform. More precisely, the Invoker will orchestrate the life-cycle of the *containerised compute runtimes* in which functions are executed in isolation (see Section 2.4). The Invoker also offers an additional isolation level for function execution and increases resilience, as multiple instances may co-exist. Moreover, special-purpose Invokers can be deployed and configured to handle the execution of functions with distinct requirements such as GPU access (see Figure 3.10).

With the proposed architecture, there is no need to follow the common practice of deploying multiple virtual machines or containers to render MEC applications resilient and responsive against downtime or bursts of workload. Instead, the event-based allocation provides inherent resilience and scalability, as fresh (or prewarmed [104]) containers can be promptly spawned and the number of running functions matches the trigger rate (as long as resources are available to the deployed invokers).

It is also important to notice that most of the Serverless MEC Platform components are shared among functions. The highly shared nature and the automated management of the whole platform have two major advantages: (i) it boosts the efficiency of resource utilisation and thereby the scalability of the Serverless MEC Platform; (ii) it allow any deployed function to be quickly scaled-out to unexpected bursts in the workload and scaled-in when the workload decreases or ceases to exist.

Empowered by the Serverless MEC Architecture, each MEC node handles the execution of several functions, and distinct client applications may rely on the same functions (e.g., those involving image processing and machine learning). Many of these functions are already supported by libraries integrated into major vendors' serverless frameworks, such as IBM Visual

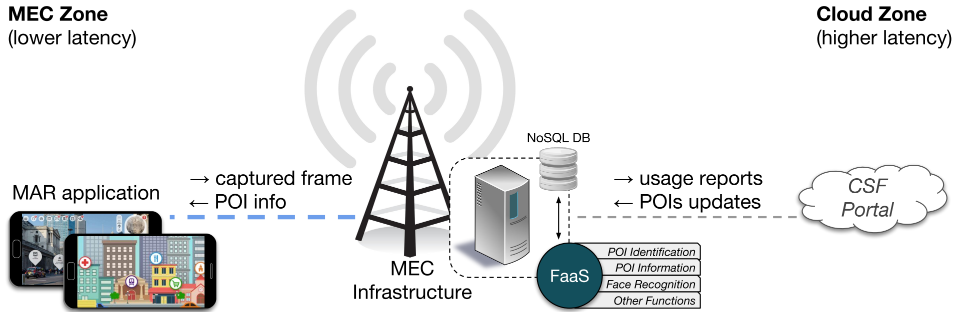


Figure 3.11: *Left: MAR application running on mobile devices. Middle: Serverless MEC Platform deployed in a surrogate MEC node. Right: cloud-based services for retrieving POI information and updating the cached POI dataset at different MEC nodes*

Recognition[5], Azure Visual Cognitive Services[6] and AWS Rekognition[7]. Similar compute runtime environments and functions are likely to become part of a Serverless MEC Platform, as the anticipation of data processing and analysis constitutes one of the primary goals of edge computing [77].

Cloud Integration

The admission and deployment of functions to the Serverless MEC Platform is performed by third-party administrators (or developers) using the *CSF Portal* from the MEC System Level in ETSI's Reference Architecture [105].

Differently from MEC Applications [100, 103], for which instantiation and termination may be requested through the CFS Portal, function deployment is limited to the upload of its source code and dependencies to the Serverless MEC Platform, which will create instances on time, accordingly to the allocation strategy discussed in this chapter.

While ETSI does not provide further details on the implementation of the CSF Portal component, one possible solution is to expose a RESTful API for handling the creation, reading, update, and deletion (CRUD) of functions. Similarly to cloud-based FaaS platforms [4, 34], such API would be accessed manually through a web portal or CLI and libraries supporting Operations Automation (Ops Automation).

[5]https://console.ng.bluemix.net/catalogue/services/watson_vision_combined
[6]https://azure.microsoft.com/en-us/services/cognitive-services
[7]https://aws.amazon.com/rekognition/

3.4.2 Platform Deployment

Access Level MEC Deployment

Latency-sensitive functions like the POI identification in the MAR are strong candidates for being offloaded to Serverless MEC Platforms hosted by MEC nodes co-located with *access-level cellular infrastructure*, namely base stations [61, 105]. In this deployment configuration, mobile devices hosting client applications are at no more than a few hops from MEC nodes, which prevents network latency, jitter and maximises throughput.

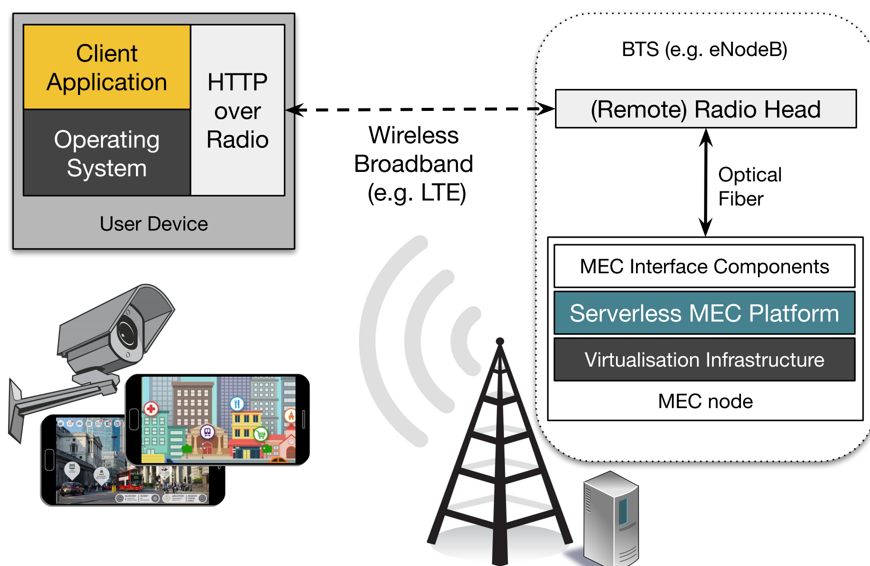


Figure 3.12: *Serverless MEC Architecture overview. Low latency applications running on user devices send requests to the Serverless MEC Platform hosted by a MEC node co-located with a BTS, which composes the mobile network infrastructure.*

Figure 3.12 provides an overview of the access-level deployment configuration. Its main physical elements are mobile devices and the MEC node. Consumer devices may be of any type running a latency-sensitive and data-intensive application that needs offloading part of its computation to a more robust platform. Hosted applications send the data to be processed (i.e., captured scene frames) to the platform through standardised protocols [109] over the radio access network (e.g. E-UTRA) using wireless broadband technology (e.g. LTE). The (remote) Radio Head (i.e. cellular antenna) composing the Base Transceiver Station (BTS) [8] bridges mobile

[8]Different wireless mobile networks generations use distinct terminologies (e.g., eNodeB in 4G and gNB in the more recent 5G).

devices and MEC nodes hosting the Serverless MEC Platform. Depending on the architecture, each BTS may serve one or multiple radio heads.

A local *Domain Name Server* (DNS) in ETSI's Reference Architecture distinguishes between requests to the RESTful platform endpoints and any other (Internet) endpoints. The main difference from a regular DNS is locality, as requests must be handled by the MEC node on the current base station To this end, the names of edge resources must be resolved locally without being propagated to public DNS servers. As a result, function invocations are handled transparently.

While the previous set-up is ideal from the network perspective, the platform performance will also depend on the MEC node's capabilities. For instance, image processing functions will significantly benefit from functionalities offered by GPUs. Hardware accelerators are also expected to be part of edge infrastructure [77]. Indeed, they are key to enable the processing and analysis of large volumes of data generated by the plethora of devices and applications at the edge.

Alternative MEC Deployments

Additional MEC deployment configurations can be found in the literature [61]. For instance, MobiScud [120] places servers in distributed data centres within or in proximity with the radio access network infrastructure; servers are accessed through Software Defined Networks. A similar approach is followed by CONCERT [56]. The latter combines the heterogeneous capabilities of servers placed at different hierarchical levels.

ETSI also foresees the possibility of harnessing the capabilities provided by Virtual Network Functions, which are expected to compose telecommunications operators' infrastructure [102]. In this alternative deployment, various node level components —including those of a Serverless MEC Platform— appear as virtualised network functions for the underlying *Network Virtualisation Infrastructure*.

Distinct MEC deployment configurations result in heterogeneous computing and storage capabilities. We argue that, regardless of the deployment configuration, the decentralised nature of the various MEC architectures in the literature imposes limitations on the capabilities of individual nodes. Hence, the efficient management of MEC node resources enabled by the Serverless MEC Architecture is critical for the feasibility of MEC and the scalability of the applications relying upon MEC services.

3.5 Proactive Recovery Protocol

3.5.1 Overview

ETSI specifies an Mp3 reference point (see Figure 3.9) for control communication between surrogate MEC nodes (e.g. co-located with neighbour base stations). This channel allows MEC platform to coordinate the relocation of MEC Application instances one from another [103].

Although the proposed Serverless MEC Architecture aims to improve the efficiency at each platform, inter-platform coordination is vital to improve scalability and resilience in the advent of two adverse scenarios: *MEC node overloading* and *platform failures*.

To tackle the previous scenarios, we propose a *Proactive Recovery Protocol*. The protocol works by enabling surrogate MEC nodes to pro-actively establish *recovery bounds*. At runtime, each MEC node monitors and advertises its state, which is then used to diverge the workload using previously established recovery bounds meanwhile the normal operation of the platform is restored.

3.5.2 Platform States

The MEC Platform Manager at each MEC node is responsible for gathering the platform state and metrics needed to infer its state. More specifically, it queries the Virtualisation Infrastructure Manager (see Figure 3.9) for the CPU and memory utilisation of the virtual machine(s) hosting an Invoker. These data are added to the *response time* and the *timeout error count* metrics collected from the Serverless MEC Platform. The collected data are then used to infer the platform state as *healthy*, *overloaded*, or *unavailable*.

Overloaded State

The Serverless MEC Platform will attempt to match demand by promptly allocating fresh containers until all Invokers reach their memory capacity. As the workload continues to increase, function activations are queued, eventually resulting in activation timeout errors. The Serverless MEC Platform is therefore considered *overloaded* if the following predicate holds:

$$(CPU \geq 90\% \vee Memory \geq 95\%) \; \wedge \; TimeoutErrorCount \geq 3$$

The first condition asserts that CPU or memory are currently at a high usage level. These are not sufficient conditions for attesting the overloaded

state, as the platform may still be able to keep response time within acceptable levels. However, they do indicate that the platform's ability to scale-out (`memory` $\geq 90\%$) or that CPU parallelism (`CPU` $\geq$ `95%`) have been reached [9]. Note that, depending on the memory quota allocated for different functions, used memory may never reach `100%`. The additional clause (`TimeoutErrorCount` $\geq$ `3`) provides the MEC Platform Manager with the last piece of evidence of the platform overload.

Unavailable State

The Serverless MEC Platform could fail due to various reasons, e.g. failures in the underlying hardware, Virtualisation Infrastructure, or in platform's components necessary to its operation. Whilst redundancy and other fault tolerance techniques may be employed to reduce the changes of faults to become failures [7], an eventual platform failure would be perceived by the MEC Platform Manager through: a notification from the Virtualisation Infrastructure Manager (*VirtualisationError*); or a sequence of function activation errors (e.g. activation timeout).

Accordingly, the platform is considered *unavailable* if:

$$VirtualisationError \vee TimeoutErrorCount \geq 10$$

It is worth mentioning that the previous conditions are not intended to be final and may be refined to reflect distinct MEC deployment configurations and service-level agreements. Nonetheless, they are in-line with the rules used for triggering auto-scaling (e.g., of virtual machines) in IaaS deployments and for determining that a distributed component is unavailable [60].

3.5.3 Proactive Recovery Bounds

We extended the role of the control channel in ETSI's Reference Architecture to allow platform state to be shared among surrogate platforms. The resulting inter-platform awareness allows the *MEC Platform Manager* to react to abnormal platform states by diverging workload to healthy platforms. Achieving this requires two further steps: (i) deciding which surrogate platform will receive the platform's workload meanwhile it is overloaded or unavailable; and (ii) changing the platform's traffic rules.

The decision of which surrogate MEC node should handle offloaded workload may follow different strategies, e.g. to take into account dynamic information such as the average service time for different functions, or how

[9] In both cases, metrics refer to the average from the virtual machine(s) that host the platform's Invoker(s).

many containers each surrogate platform is currently using (or retaining idle). Another aspect regards the distribution of control, which may be centralised (e.g. a cloud-based orchestrator), partially decentralised (e.g. elected leaders within MEC clusters [93]), or fully decentralised (e.g. MEC nodes behave as self-organizing agents [23]).

Similarly to the internal load balancing among invokers discussed earlier in this chapter, the highly elastic and dynamic nature of resource allocation in FaaS makes it harder to keep track of the platform's internal state. An effective recovery solution must consider this volatility and avoid taking decisions based on information that becomes rapidly outdated. In the other hand, a minimum coordination is required to prevent conflicting decisions that could further jeopardize the stability of the MEC system.

In light of this, our recovery protocol enables MEC nodes to pro-actively form *recovery bonds* with their neighbours. To cope with simultaneous overloading or malfunctioning of two or more surrogate platforms, each MEC node establishes recovery bounds of different *degrees*, which are activated upon overloading or unavailability of the source Serverless MEC Platform in a predetermined order and accordingly with the state of the target platform. Simultaneously, the protocol aims to distribute the recovery bounds among nodes in a balanced manner to prevent disrupting their own operation in case multiple nodes become overloaded or unavailable.

The Proactive Recovery Protocol is designed to operate without intervention from an external orchestrator or cluster leaders. Moving away from centralised control improves the robustness and scalability of the recovery solution [23, 48]. While cluster-based solutions may achieve better results within the cluster space, global performance will depend on the clusterisation algorithm itself. For instance, if clusters operate in isolation, neighbour nodes in distinct clusters may loose the opportunity to cooperate. Also, leaders may become central point of failure within their cluster.

The proposed protocol is carried out by ETSI's node level MEC Platform Manager and comprises the following messages:

- **`ASK(NODE_ID, DEGREE, PRIORITY)`**. Indicates a target MEC node that `NODE_ID` is willing to form a recovery bound.
  - The `DEGREE` parameter defines the order in which the target node will be looked upon for recovery by the requesting node.
  - The `PRIORITY` parameter refers to the criticality of the recovery bound of a given degree; it provides criteria for solving conflicting requests (i.e., overlapping recovery bounds).

  - `OK|NOK` informs the source node about the outcome of its recovery bound creation request.

- **`UPD(NODE_ID`$\rightarrow$`(DEGREE:PRIORITY))`**. Used by target nodes to inform surrogate nodes about a new bound of `DEGREE:PRIORITY`.

The protocol makes no assumptions about the order in which messages are sent and received, which is a critical requirement in distributed systems. Furthermore, MEC nodes may activate the protocol at any time, for example when new MEC nodes are added to an existing MEC system.

The protocol operates at runtime. Nonetheless, it may be refined into two phases: *initialisation* and *adaptation*. At initialisation, the node will attempt to form recovery bounds of different degrees. Whenever new MEC nodes join the system or become unavailable, *adaptation* takes place with the self-organisation of recovery bounds until a new equilibrium is achieved.

Listings 1,2,3, and 4 describe the procedures implementing the recovery protocol. The `initialiseBounds` procedure is used at initialisation. It receives two parameters: the source node (containing surrogate nodes); and the maximum degree. The latter parameter is useful for limiting the protocol complexity, i.e., the number of messages exchanged and the time needed to converge to a stable solution. In contrast, the higher the maximum degree, the more alternatives a MEC node will have for recovery.

The `initialiseBounds` procedure will solve one recovery bound at a time, starting with the most relevant degree (i.e., `DEGREE=1`) up to the maximum degree (by default, the number of surrogate nodes). For each degree, the `checkBound` procedure is called.

The `checkBound` procedure randomly iterates over surrogate nodes using a hash function to mitigate collisions. The iteration will stop at the first node that: (i) has no bounds of lower degrees with that source node —which otherwise would defeat the purpose of having multiple bounds; and (ii) has less or, at most, equal number of bounds of that specific degree with other nodes —which steers the system towards a balanced solution.

The selected node is then sent an ASK message with the following `PRIORITY`: one (1), if there are other surrogate node(s) without bounds for that particular degree that have not been asked for; two (2), if all other

Algorithm 1 initialiseBounds($node, maxDegree$)

```
1: for degree ← 1 to maxDegree do
2:     CHECKBOUND(node, degree)
3:     WAITFORDEGREE(node, degree)
4: end for
```

Algorithm 2 checkBound($node, degree$)

```
1: surrogateNodes ← node.surrogateNodes
2: minBoundCount ← GETMINBOUNDCOUNT(surrogateNodes, degree)
3: altNodes ← GETALTNODES(surrogateNodes, degree, minBoundCount)
4: curBound ← GETCURRENTBOUND(node, degree)
5: while (targetNode ← NEXTNODE(surrogateNodes, degree)) not NULL do
6:     targetBounds ← GETBOUNDS(targetNode, degree)
7:     hasLowerDegree ← HASLOWERDEGREE(node, targetNode, degree)
8:     if targetBounds.size = minBoundCount and not hasLowerDegree then
9:         priority ← 1
10:        minID ← GETMINID(targetBounds)
11:        if curBound not NULL and curBound.targetNode = targetNode then
12:            if altNodes.size > 1 and node.id ≤ minId then
13:                priority ← 2
14:            else if altNodes.size > 1 and node.id > minId then
15:                altNodes ← REMOVEFROM(altNodes, targetNode)
16:                CONTINUE
17:            else
18:                priority ← 3
19:            end if
20:        end if
21:        ok ← ASKFORBOUND(sourceNode, targetNode, degree, priority)
22:        if ok then
23:            if curBound not NULL and priority = 1 then
24:                DROPBOUND(curBound)
25:            end if
26:            BREAK
27:        end if
28:    end if
29: end while
```

Algorithm 3 handleASK($node, sourceNode, degree, priority$)

```
1: maxPriority ← GETMAXPRIORITY(degree)
2: if priority < maxPriority then
3:     REPLYNOK()
4: else
5:     bound ← GETBOUND(sourceNode, node)        ▷ Any degree and priority
6:     if bound not NULL then
7:         REMOVEBOUND(bound)
8:     end if
9:     newBound ← CREATEBOUND(sourceNode, node, degree, priority)
10:    MULTICASTUPD(node, newBound)
11:    REPLYOK()
12: end if
```

Algorithm 4 handleUPD($node, targetdNode, newBound$)

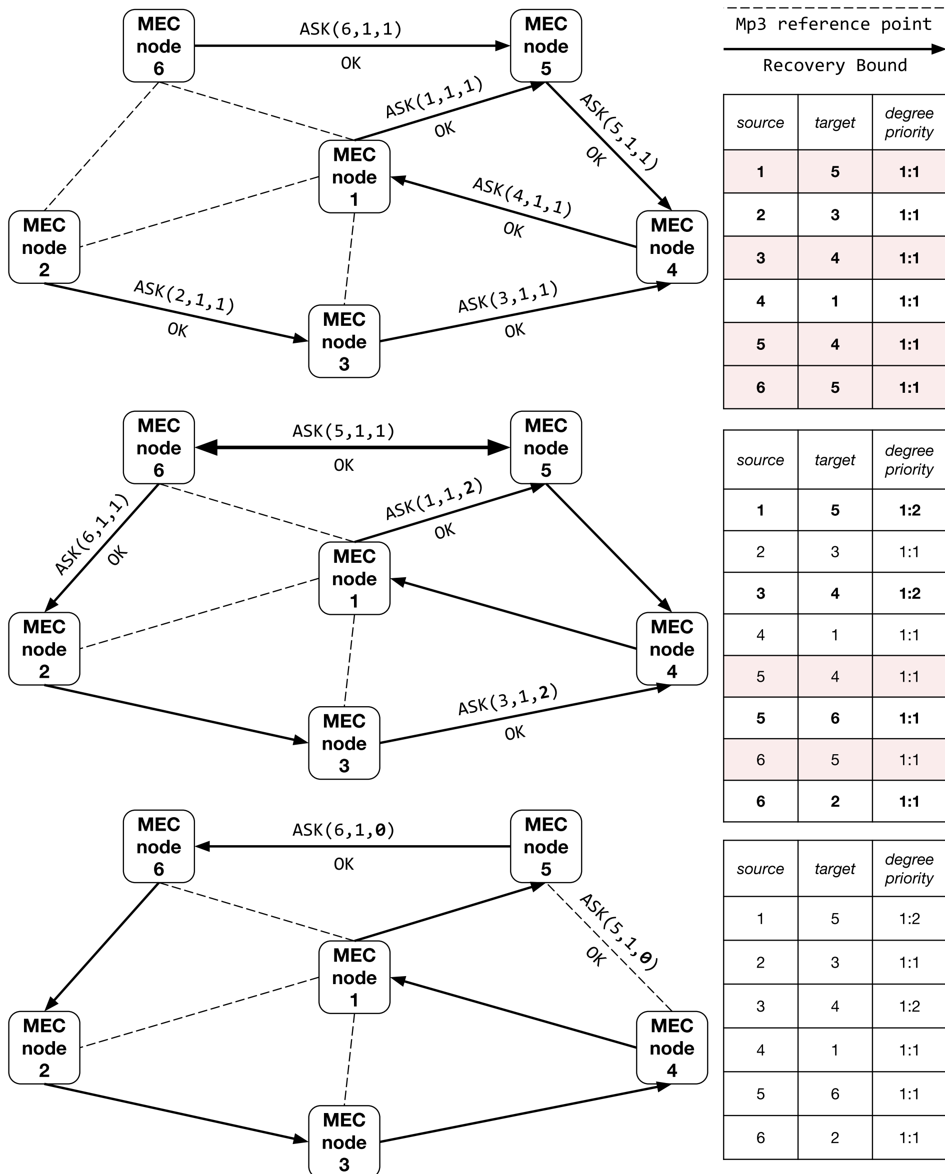
```
1: bound ← GETBOUND(node, targetdNode)
2: ADDBOUND(targetdNode, newBound)
3: if bound not NULL then
4:     if bound.degree > newBound.degree then
5:         CHECKBOUND(node, bound.degree)        ▷ Higher degree bound overridden
6:     end if
7: else
8:     CHECKBOUND(node, newBound.degree)         ▷ New overlapping bound
9: end if
```

surrogate nodes have an equal or greater number of bounds; and three (3), if this is the only remaining alternative (e.g. MEC nodes at more remote areas). In case of draw, the node disregards the current target if alternatives exist *and* the node's unique ID *is not minimal* among contenders.

Upon receipt (`handleASK` procedure), the target node checks for the existence of recovery bounds for that same degree. It refrains from creating a new bound only if a *higher priority bound* is found (`NOK` reply); otherwise a recovery bound is created with the requested degree and priority (`OK` reply). For each created bound, the target node updates its internal state and multicast this event to surrogate nodes (`UPD` message).

To handle situations in which multiple nodes ask for the same degree and priority, nodes react to `UPD` messages (`handleUPD`) by checking if it affects its own recovery bounds. If the updated state contains additional nodes sharing the same degree and priority, the node ID is used as tiebreaker criterion; those with higher ID will repeat the iteration over its surrogate nodes and look for the first node with more favourable state (i.e., that has fewer bounds for that particular degree). If a better alternative is found, it attempts to create a new bound with the same degree and priority; if successful, it will drop the previous bound by sending a new `ASK` message with the same degree, but `PRIORITY=0`. Conversely, if no better alternative is found, the node will enforce the existing recovery bound by sending a new `ASK` message, this time with `PRIORITY=2` —if alternatives as good as this still exist— or `PRIORITY=3` —if this is the single or last alternative.

It is important to note that other nodes in similar situation (i.e., with overlapping bounds) will perform the same procedure. The priority mechanism allows nodes in more adverse scenarios (e.g. exhausted alternatives) to signalise their situation. Other nodes will use this information to reorganise their bounds whenever alternatives are available. If the MEC topology is balanced (i.e., equal number of surrogate nodes), at least one of the nodes



source	target	degree priority
1	5	1:1
2	3	1:1
3	4	1:1
4	1	1:1
5	4	1:1
6	5	1:1

source	target	degree priority
1	5	1:2
2	3	1:1
3	4	1:2
4	1	1:1
5	4	1:1
5	6	1:1
6	5	1:1
6	2	1:1

source	target	degree priority
1	5	1:2
2	3	1:1
3	4	1:2
4	1	1:1
5	6	1:1
6	2	1:1

Figure 3.13: *Recovery bounds formation. A mesh of 6 MEC nodes form balanced recovery bounds of degree 1 (subsequent degrees are omitted).*

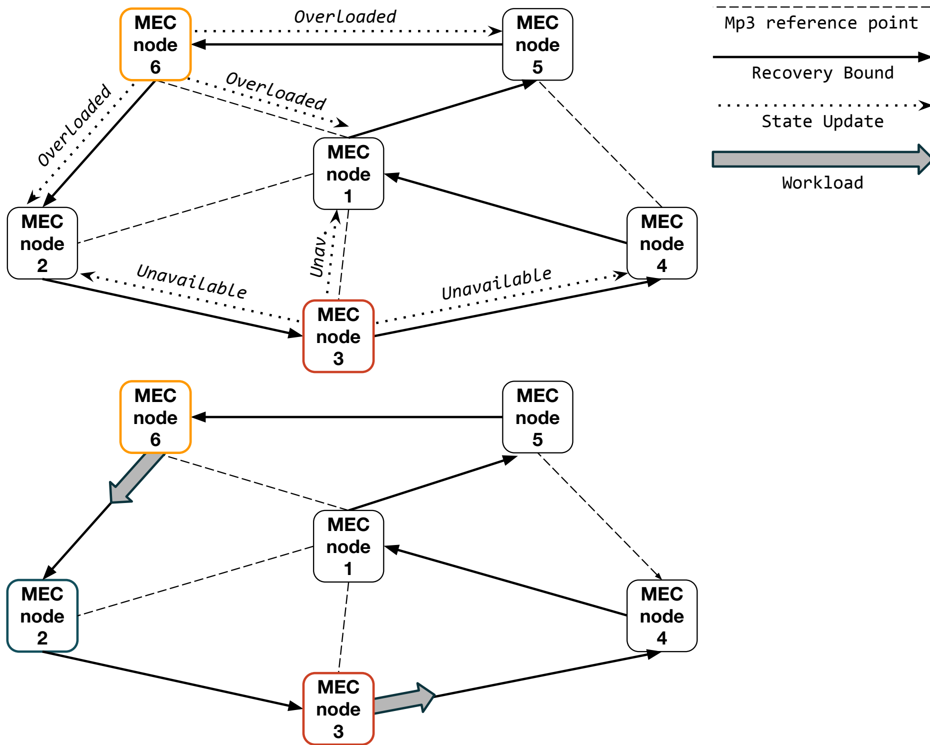


Figure 3.14: *Proactive Recovery Protocol in action. Nodes 3 and 6 advertise their abnormal state to surrogate nodes. Workload is diverged from node 6 to node 2, whereas MEC node 3 offloads its workload to MEC node 4.*

is likely to find a better alternative and drop the disputed recovery bound. Also, the protocol assures that the recovery bounds of different degrees are always established with distinct targets. The protocol will converge when: (i) all aimed degrees have been satisfied; (ii) no better alternatives can be found by any of the MEC nodes and for all aimed degrees.

Figure 3.13 depicts the recovery protocol in action with a reduced MEC topology composed of 6 nodes. For the sake of simplicity, only the formation of the first degree is depicted. After randomly choosing their first degree targets, pairs `(1, 6)` and `(3, 5)` establish overlapping bounds with nodes `5` and `4` respectively. Higher ID nodes (i.e., `5` and `6`) will establish new bounds with other nodes in better conditions and drop their previous bound; in turn, nodes `1` and `3` will enforce their bounds (`PRIORITY=2`).

3.5.4 Recovery in Action

Once recovery bounds have been pro-actively established, the MEC Platform Manager is able to diverge workload in case of abnormal platform state. The Proactive Recovery Protocol relies upon minimal information regarding the platform's internal state, which is analysed locally by the MEC Platform Manger and shared among surrogate nodes (as inferred *platform states*), accordingly to the Platform State Monitoring introduced in Section 3.5.2. The adopted approach prevents excessive changes to the offloading decision and consequently mitigates overhead (e.g. in changing the platform's traffic rules). More importantly, it reduces the odds of enacting conflicting decisions with other MEC nodes.

At runtime, the MEC Platform Manager monitors and advertises the state of its own platform and listens for events coming from surrogate nodes. Upon detecting an abnormal state of its own platform, the MEC Platform Manager proceeds as follows:

- If the Serverless MEC Platform is *overloaded*, it chooses the recovery bound of *lowest degree* whose target platform is in *healthy* state.
  - If none is found (e.g. due to generalised overloading), it refrains from diverging workload.
- If the Serverless MEC Platform is *unavailable*, it chooses the recovery bound of *lowest degree* whose target platform is in *healthy* state.
  - If none is found, it chooses the *lowest degree* whose platform is *overloaded*.
  - In the worse case scenario in which none is available (e.g. generalised catastrophic failure), it refrains from diverging workload.

Figure 3.14 illustrates the recovery procedure with a scenario in which 2 out of 6 MEC nodes are in abnormal states.

CHAPTER 4

Mobile-Edge-Cloud Continuum Through Serverless and Autonomic Computing

4.1 Overview

In Chapter 3 we proposed a Serverless MEC Architecture for enabling latency-sensitive and data-intensive applications to exploit the services provided by surrogate MEC nodes. In this section, we consider a broader range of deployment configurations resulting from the combination of mobile, edge, and cloud computing, from now on referred to as the *Mobile-Edge-Cloud Continuum*, the *Compute Continuum*, or just *Continuum*.

While MEC is one of the most mature edge-centric architecture, other models also received considerable attention from both industry and researchers. This is particularly so for two alternative concepts, namely cloudlets [96], which encompasses private deployments of trusted, powerful computers or cluster of computers; and fog computing [18], whose multi-layered architecture stretches from densely distributed devices at the network edge all the way to cloud data centres.

From a different angle, other authors [27,116,133] consider end-user devices such as smartphones, tablets, and IoT (referred to as *edge devices*) not as just consumers, but also as service providers. In this view, the primary

goals of edge computing are fulfilled through the direct cooperation and collaboration among edge devices, which execute locally tasks that would otherwise be performed by services deployed to cloud infrastructure.

The heterogeneity among the existing edge-centric architectures is profound and multi-faceted. Nonetheless, different strands share the objective of paving the way for new and disrupting applications empowered by seamless connectivity and pervasive computing technology. These applications and the edge paradigm as a whole would benefit from the convergence provided by a unified and comprehensive model.

Extending the Serverless MEC Architecture in Chapter 3, we present a unified Mobile-Edge-Cloud Continuum model and propose A3-E, a framework for tackling the management of the life-cycle of serverless functions in the Continuum. A3-E framework exploits both serverless [9, 87] and autonomic computing [47] to allow stateless and lightweight functions to be opportunistically and efficiently fetched, deployed and exposed as remote services by heterogeneous cloud and edge providers or consumed locally. Additionally, A3-E enables mobile applications to adapt to scenarios in which surrogate edge servers are not available and contexts in which performing computation locally is more advantageous than offloading it.

The remainder of this chapter is organised as follows. In Section 4.2, we discuss the challenges of developing mobile applications for the Mobile-Edge-Cloud Continuum and motivates it with a Running Example. Section 4.3 introduces the A3-E framework and its main components. Section 4.4 provides a detailed description of the A3-E framework, whereas Sections 4.5 and 4.6 introduces prototype implementations.

4.2 The Mobile-Edge-Cloud Continuum

The conjunction of mobile-, edge-, and cloud-computing have the potential to form a compute continuum on which new and disruptive types of applications can be built. This continuum enables the seamless convergence of heterogeneous infrastructures, stretching all the way from cloud data centres down to mobile and IoT devices, including intermediate steps such as ISP gateways, cellular base stations, and private deployments.

Given that the two fundamental elements of computation are data and behaviour, the first challenge of developing an application that exploits the Mobile-Edge-Cloud Continuum consists in deciding where in the Continuum application data and behaviour should be deployed.

If we focus on behaviour, it is common to distinguish between stateful and stateless computation. The main distinguishing factor between state-

ful and stateless components is that the latter does not retain information from one invocation to the next, nor produce internal side-effects. Also, the produced output depends solely on the received inputs.

If we focus on data, it is common to distinguish between mutable and immutable data. While mutable data can be modified after its creation, immutable data is initialised once and for all at deployment time, i.e., immutable data remains static throughout the life-cycle of the component until a new deployment may update it.

The choice of where to deploy data and behaviour needs to be informed by the heterogeneity that exists between the different infrastructures that constitute the Continuum. Amongst other things, one must take into account important QoS aspects, such as availability, consistency, and latency.

Data and behaviour that are deployed to cloud solutions will benefit from high availability, at the cost of introducing higher latency. In contrast, components deployed to densely distributed edge nodes will benefit from minimal latency and maximal throughput, at the cost of more constrained resources and therefore reduced availability (e.g., whenever edge nodes in proximity are overloaded). Moreover, the consistency of data hosted by geographically distributed components is much harder to achieve.

A third possibility is to privilege mobile and IoT device own resources. However, this could lead to significant battery drains and performance degradation. Balancing all these trade-offs at design time may be infeasible, especially in volatile contexts of operation. The critical challenge is, therefore, to allow applications to dynamically and opportunistically decide where data and behaviour should be deployed and executed.

As we shall present in Section 4.3, it is our view that the Compute Continuum should focus on stateless computation —i.e., *functions*. Stateless components hosting immutable data are much easier to replicate (and test) across the continuum since no data synchronisation is required and any data needed by the computation can be obtained at deployment time. Nevertheless, stateful computation and mutable data cannot be excluded from most modern applications. For these cases, we envision a mixed solution in which typical mobile and cloud computing architectures are adopted alongside the Mobile-Edge-Cloud Continuum.

Developing applications in the Compute Continuum also poses other essential challenges. For example, when selecting where a specific computation should be achieved, we need to take into account security aspects, such as authorisation, confidentiality, and integrity. Also, deployment and execution in the Continuum will need tool support, e.g., for performing modelling, analysis, verification and integration testing. Although these

challenges are interesting and relevant research endeavours in themselves, in this chapter we focus on establishing the basis upon which the Compute Continuum can develop. Specifically, we focus on achieving: (i) the effective deployment of serverless functions by the heterogeneous infrastructures that constitute the Continuum; and (ii) the dynamic selection of where to execute each function while satisfying application requirements.

4.2.1 Running Example

In this section, we motivate the Mobile-Edge-Cloud Continuum with an example scenario that involves multiple applications that rely on functions executed in the cloud, edge, in the user's mobile device.

First, let us consider the MAR application introduced in Chapter 3 and employed by our user to explore POI in the city she is currently visiting. The MAR application relies on computation-intensive and latency-sensitive tasks; as such it can benefit from adopting the Continuum. Indeed, despite the capacity of mobile devices, it is common with these kinds of applications for users to experience functional and non-functional degradation (e.g. low refresh rate, battery drain). By offloading the POI identification function to a more capable server, the user may enjoy an improved Quality-of-Experience. While Serverless MEC Platforms are ideal candidates, they may not always be available in the user location. Tourists visiting less popular areas could still rely on cloud-based serverless platforms if latency is acceptable. Alternatively, developers may opt to deploy a reduced POI dataset alongside the mobile application for covering these specific areas.

After her tour, our user calls for an Autonomous Vehicle to drive her back to the hotel. During the journey, she starts editing the pictures shot during the day. This activity is only interrupted by notifications from the (still running) MAR application providing nearby POI information. To reduce the processing time and avoid battery drain, the vehicle features an edge server that provides a dynamic catalogue of functions, including those needed by the MAR application. The vehicle itself relies on a *route planning* service to calculate the best plan to reach the destination. Since latency is not a critical requirement, the latter is served by a cloud provider.

Once at her hotel room, our user continues to edit the images and videos that she shot during the day. Later on, she decides to enjoy a Mobile Game. The game relies on complex calculations that pose a burden to her mobile device. To support guest applications, the hotel provides an edge server equipped with GPU. The server identifies the Image Editing and Mobile Game applications and, after fetching and installing the necessary software,

Figure 4.1: *The conceptual model for the A3-E framework and the Compute Continuum.*

starts providing the required functionality. Services managing game scores and player authentication are hosted by cloud data centres.

4.3 System Model

The conceptual model depicted in Figure 4.1 defines the entities composing the Mobile-Edge-Cloud Continuum and their association with the A3-E framework. Next, we introduce each of these entities. In particular, the AE-E Framework —responsible for the self-management of the life-cycle of serverless functions— is presented in details along Section 4.4.

4.3.1 Infrastructure Model

The Mobile-Edge-Cloud Continuum is constituted of multiple domains. A domain yields a common abstraction of the heterogeneous compute, storage and network resources that make up the Continuum, e.g. physical servers and virtual machines, storage, and network infrastructure providing access to the provided services. In the particular case of MEC, the term domain generalises the ETSI concept of a *MEC node* [105].

Remote Domains

At the topmost level of the Compute Continuum are *cloud domains*, which are accessible through and identified by their public Internet names. Fulfilling the gap between cloud domains and client devices are *edge domains*. In our framework, cloud and edge domains are also referred to as *remote domains*. Edge domains are further distinguished by the networking technol-

ogy they use: *mobile-edge domains* (in the context of MEC) are co-located with cellular infrastructure and accessed through mobile network technology (see Chapter 3). In contrast, *local-edge domains* integrate with wireless local area networks (e.g., a public or private WiFi Gateway).

Our vision of a cloud domain is aligned with the concept of *multi-clouds* [88], in which client applications rely on distinct providers for the same functionality. Moreover, cloud domains are aligned with the concept of *regions* adopted by major cloud providers. As the name suggests, a region (or *zone*) refers to the geographical area where independent cloud data centres from the same provider are located. The deployment of applications to multiple regions is typically used for fault tolerance (e.g. in the advent of catastrophic events), load balancing, and to improve network performance —the primary motivation in the context of our work.

Densely distributed edge domains further refine the concept of regions. Similarly to multi-clouds, we assume the existence of multiple edge providers. For example, *AWS Lambda*, *Google Cloud Functions*, *IBM Cloud Functions*, *Microsoft Azure Functions* are seen as distinct cloud domains; different MEC operators are seen as distinct mobile-edge domains; whereas public and private deployments (e.g. cloudlets [97]) in the same area consist of alternative local-edge domains.

Mobile Domains

In opposition to remote domains are local *mobile domains*. In the Continuum, mobile devices play two roles: they are both *consumers* and *providers*. As consumers, mobile devices have access to compute services provided by edge and cloud domains. More precisely, mobile devices consume *Function-as-a-Service* from edge and cloud domains. Additionally, mobile devices provide resources for the execution of functions required by the applications they host.

The motivation for including the resources and capabilities from mobile devices in the Compute Continuum is threefold. First, due to the substantial increase in computational capacity exhibited by modern devices [84, 118]. Second, due to the ability of these devices in executing serverless functions. Third, to give applications a zero network latency and highly available alternative to remote domains. Also importantly, this choice is in-line with the edge computing vision [116] in which more powerful mobile and IoT devices play the role service providers.

Figure 4.2 depicts the continuum model from a deployment perspective. In this representation, a single mobile device hosts many continuum applications, which harness (through the *Mobile Middleware* introduced later on

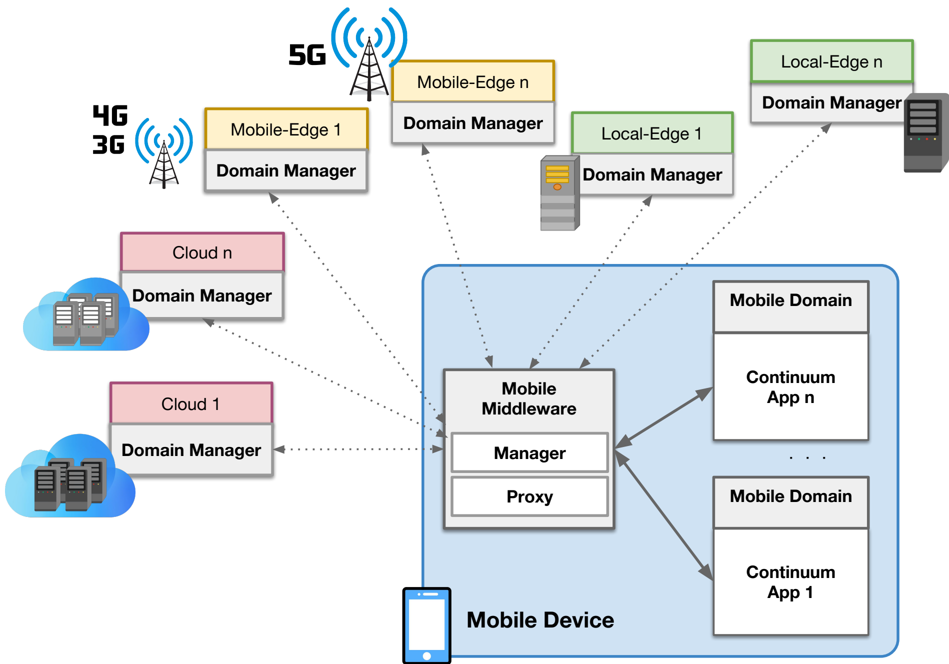


Figure 4.2: *High-level architecture of the Compute Continuum. A single mobile device interacts with its (local) mobile domain, and with distinct edge and cloud domains.*

in this section) the compute services provided by its (local) mobile domain and distinct edge and cloud domains.

4.3.2 Continuum Functions and Requirements

Among the entities composing our framework are *functions* —executed by various continuum domains— and accompanying *requirements*.

Continuum functions are aligned with the FaaS model and are specified with the same attributes described in Section 2.4 (i.e., memory capacity, runtime, timeout, and concurrency). Additionally, serverless functions are specified with two types of requirements:

- **Location-Requirements**: `LOCAL-ONLY`, `REMOTE-ONLY`, `EDGE-ONLY`, `LOCAL-EDGE-ONLY`, `MOBILE-EDGE-ONLY`, `CLOUD-ONLY`.
- **QoS-Requirements**: `COST`, `BATTERY-CONSUMPTION`, `RESPONSE-TIME`.

Location-Requirements enable application developers to restrict where

in the continuum a function may be executed. The rationale behind this requirement is to give application developers the final word on which type of domain the application relies upon whenever this decision may or must be taken at design-time. Some of the use cases are: to prevent the overloading of resource-constrained devices with compute-intensive functions; to prevent the use of cloud domains with data-intensive and latency-sensitive functions; or even to condition the use of more exclusive domains (e.g. mobile-edge domains provided by telecom operators) to specific conditions external to the context of operation (e.g. premium user accounts).

In contrast, *QoS-Requirements* enables developers (and, as we shall discuss in Section 4.5, end-users) to delegate the offloading decision to a *managing* [47] component through the specification of *Quality-of-Service* attributes. QoS-Requirements are composed of *constraints* and *weights* for one or more QoS attributes. The latter is then monitored and analysed at runtime and serve as input for the multi-attribute decision of which domain should host the execution of each required function in different contexts of operation.

4.3.3 Continuum Application

The architecture of a Continuum Application consists of:

- *stateless application logic* and *immutable data* forming serverless functions, which are dynamically deployed to mobile, edge, and cloud domains;
- *stateful components*, which may still be needed in an application, are deployed to cloud data centres or the mobile device, depending on design-time decisions; and
- typical client-side components such as application logic handling internal and external events, and those responsible for user interfaces.

Continuum applications leverage the serverless architecture proposed in Chapter 3 to prevent data consistency problems that would arise if stateful components containing transient or persistent state were deployed to finely-distributed edge nodes. The adopted model also allows function instances to co-exist along the continuum. More importantly, functions may be deployed and undeployed independently without the need for state migration, favouring the seamless transition from one continuum domain to another. These benefits are particularly important to cope with user mobility and, as discussed in Chapter 3, with unpredictable workload fluctuations.

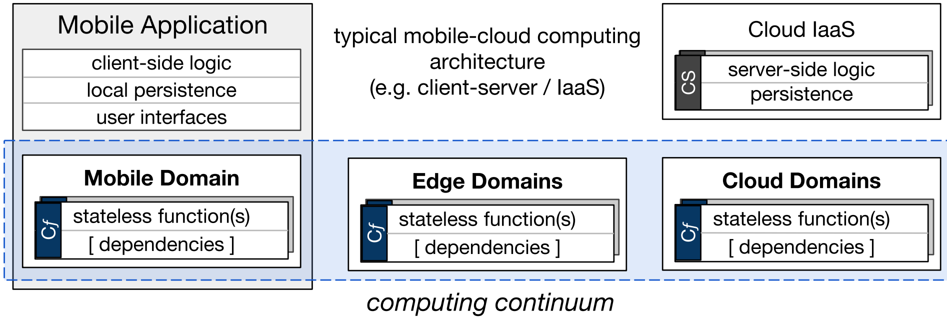


Figure 4.3: *Architecture of an application exploiting both the Compute Continuum —through functions (Cf) executed by mobile, edge and cloud domains— and typical mobile/cloud architecture —through cloud services (CS) and local components.*

It is worth mentioning that we target *general purpose, application-level functions*, in contrast to *Virtual Network Functions* (VNF). While our framework does not target VNF, they are considered as a key technology supporting the infrastructure of edge domains, e.g. by routing requests to/from MEC nodes in specific deployment configurations [56, 120].

Figure 4.3 illustrates the high-level architecture of a mobile application that leverages the Continuum. Serverless functions are opportunistically deployed to different domains along the continuum. Additional components follow a conventional mobile-cloud architecture: the front-end comprises *client-side application logic* (possibly stateful), *local persistence*, and *user interfaces*; a typical IaaS/CaaS deployment hosts the back-end, which comprises *server-side application logic* and *remote persistence*.

4.3.4 Mobile Middleware and Domain Manager

Representing the two principal parts in the Compute Continuum —namely mobile devices and continuum domains— are the *Mobile Middleware* and the *Domain Manager*. Their purpose is to tackle the life-cycle of serverless functions, which is achieved through the coordinated activities defined by the A3-E Framework (as we shall describe in Section 4.3.5).

Mobile Middleware

The Mobile Middleware performs two central roles: of a *managing system*, and a *proxy*.

As a managing system, it performs activities related to the autonomic management [47, 81, 112] of continuum applications, namely:

- registering and advertising the functions required by continuum applications —*self-configuring*;
- discovering domains —*environment-awareness*;
- monitoring the QoS (e.g. response time) of functions provided by distinct domains —*environment-awareness*;
- monitoring the internal state (e.g. battery level) and user requirements (e.g. battery consumption) —*self-awareness*; and
- preventing and handling failures —*self-healing*.

More importantly, the Mobile Middleware is responsible for analysing, based on perceived QoS and requirements, the best domain to execute each function required by continuum applications hosted by the mobile device (i.e., *self-optimising*). Given the specification of multiple requirements, this choice consists of a *Multiple-Criteria Decision-Making* (MCDM) [76, 86].

The result of the MCMD is used to transparently *proxy* Continuum Application requests to the best domains, i.e., the Mobile Middleware also acts as a proxy to continuum applications.

Domain Manager

Similarly to the Mobile Middleware, the Domain Manager also plays the role of a *managing system*.

As a managing system, it provides autonomic capabilities —also known as self-* properties [47, 81, 91]— to the FaaS platform handling the execution of serverless functions, namely:

- identifying functions required by continuum applications (*environment-awareness*);
- fetching and deploying the required functions (*self-configuring*);
- monitoring the context (e.g. workload) in which the platform operates (*environment-awareness*); and
- keeping track of the FaaS platform QoS (*self-awareness*).

The Domain Manager also oversees the FaaS platform in terms of how many containers are allocated per function given its context of operation and SLA (*self-optimising*).

System-wise, the Domain Manager is responsible for advertising the domain's presence and capabilities; and for providing the Mobile Middleware with performance metrics that guide its managing decisions.

4.3.5 Life-cycle Management Problem

When managing the life-cycle of serverless functions, there are two potentially conflicting goals: i) the satisfaction of application requirements and ii) the optimisation of the resources consumed by these applications.

We focus on three kinds of application requirements: response time, battery consumption, and monetary cost. Simultaneously, we target the efficient and scalable use of computational resources —namely CPU, memory, and storage— from the cloud and edge providers.

In the continuum, a multitude of disjoint cloud and edge domains can host the execution of functions, i.e., they must be able to handle operational aspects such as *downloading* and *deploying* required functions. Also, mobile clients will freely enter and exit geographical areas that are covered by distinct edge domains; even cloud domains are likely to see considerable variations to their aggregate workload over time. In such a scenario, it may be infeasible to know a priori or accurately predict the origin and intensity of the workload that the different functions at each domain might see.

From an edge/cloud operator's perspective, an efficient and scalable allocation of the (virtualised) resources in each of its domains must be able to cope with the maximum acceptable response time of each admitted function. To be efficient, the allocation of resources must mimic the corresponding fluctuations of demand, i.e., be highly responsive. Consequently, it is essential for the mechanisms governing resource allocation in each domain to be *aware* of the actual and potential demand for each provided function.

To better express the previous problem, let us define the set of functions $F = \{f_j \mid j = 0, 1, ..., J\}$. Each $f_j \in F$ may have zero to many instances allocated. Each instance is bound to a container; the resources allocated to each container (i.e., CPU and memory) follow the requirement specification for the corresponding function, coherently to existing FaaS platforms [4,67, 104]. Vector $\bar{c} = (c_1, ..., c_J)$ holds the number of instances for each $f_j \in F$.

Now let us assume that each function $f_j \in F$ is additionally bound to an SLA specifying a *maximum accepted service time* Δ_j and a *maximum concurrence limit* C_j, so that $c_j \leq C_j$. Given the fluctuations in the workload and average service time τ_j (comprising *setup time*, *queue time*, and *execution time*) of each function, the aim of the Domain Manager is to calculate the vector $\bar{c}$ so that $\tau_j \leq \Delta_j, \; \forall f_j \in F$. In other words, the domain manager is responsible for regulating the pace in which containers are allocated regarding the workload; the more strict the service time to be met, the more responsive must be the creation of containers in the advent of workload fluctuations. Conversely, the higher the chances of resource contention, the

more conservative the allocation must be.

Another *touchpoint* that affects allocation responsiveness consists of the number of *pre-warmed* containers. As discussed later in this chapter, some FaaS platforms [104] allows its operator to statically define the number of pre-allocated containers that may promptly be loaded with any function compatible with that specific compute runtime (see Chapter 2.4). While pre-warming containers drastically reduce cold start, this approach also increases the underutilised capacity and therefore hurts allocation efficiency —crucial for densely distributed edge domains. The Domain Manager must, therefore, steer the system towards a balance between *provisioning responsiveness* and *allocation efficiency*.

From the Continuum Application's viewpoint, the challenges are mainly twofold: (i) edge domains need to be dynamically *discovered* and *informed* about application requirements; and (ii) the application may need to choose, at runtime, among different domains.

The QoS delivered by distinct domains may vary due to several reasons. Let us focus, for example, on response time. The multiple hops of networking separating cloud data centres from mobile devices may result in network latency and jitter. On the other hand, edge domains may suffer from resource contention. Mobile devices are inherently less powerful; they may also suffer from performance degradation, especially if multiple applications are simultaneously in execution.

The battery drain experienced by mobile devices also depends on the context of operation [20]. If the computation is performed locally, the CPU and GPU are the main culprits. Conversely, the burden posed by computation offloading depends mostly on the payload size and network throughput [52]. For the mobile application, an optimal domain selection is therefore the one that, given a set of requirements, satisfies the multi-criteria decision of which domain to use according to the perceived QoS of the available domains. Moreover, since serverless functions are modular and independently deployed, it consists of individual decisions regarding each function consumed by the application.

To better express the domain selection problem, let us extend the previous formulation with an application A that relies on the set of functions $F_A = \{f_{A,i} \mid i = 1, 2, ..., I\}$; and the set of disjoint continuum domains $D = \{d_p \mid p = 1, 2, ..., P\}$ covering the mobile device hosting A. Each domain $d_p \in D$ provides a set of functions $F_p = \{f_{p,j} \mid j = 1, 2, ..., J_p\}$. For each $f_{A,i} \in F_A$, there is at least one domain providing that function, that is, $\exists\ f_{p,j} \in \bigcup_{p=1}^{P} F_p \quad s.t. \quad f_{p,j} = f_{A,i},\ \forall f_{A,i} \in F_A$.

Now let us consider that each $f_{A,i} \in F_A$ is bound to a set of QoS re-

quirements $QoS_A = \{q_{A,u} \mid u = 1, 2, ..., U_A\}$ and that each $q_{A,u} \in QoS_A$ is represented by a tuple $(ct_{A,u}, w_{A,u})$ respectively defining a *constraint* (e.g. $maximum\ response\ time\ \leq 300ms$) and a *weight* for that attribute, where $0 \leq w_u \leq 1,\ w_u \in \mathbb{R}_{\geq 0}$. For each $q_{A,u} \in QoS_A$, $actual_{p,A,u}$ defines the *value* for that QoS attribute as perceived by the client. It follows that, for each $f_{A,i} \in F_A$, the goal is to select the domain $d_p \in D$ that maximizes the utility function $U_A(p) = \sum_{u=1}^{U_A} w_{A,u} * actual_{p,A,u} \quad s.t. \quad actual_{p,A,u} \vdash ct_{A,u},\ \forall q_{A,u} \in QoS_A$ (i.e., that all constraints are satisfied).

4.4 A3-E Framework

In this section we present A3-E, a framework tackling the self-management of the life-cycles of serverless functions. A3-E inherits its name from its four constituting activities, namely: ***A**wareness, **A**cquisition, **A**llocation, and **E**ngagement*.

A3-E targets the efficient and scalable placement of functions along the continuum and the satisfaction of requirements such as the response time, battery consumption, and availability. To achieve it, client devices and heterogeneous continuum domains take part in the automated and opportunistic decision of which continuum resources —among those of mobile, edge, and cloud domains— should be employed in the execution of each of the functions required by applications.

4.4.1 Overview

Figure 4.4 provides an overview of the A3-E framework. Activities are carried out by the *Domain Manager* and the *Mobile Middleware*, and coordinated asynchronously through *events*. Event notifications are depicted by solid (local event) and dashed (remote event) arrows. Event labels are depicted in Table 4.1.

Next, we describe each of the four main activities composing the A3-E framework. Note that, to address the intrinsic heterogeneity of the continuum, A3-E is flexible concerning how different domains implement its activities. Whenever appropriate, we shall highlight these differences; we shall also refer to the activities and events depicted in Figure 4.4.

4.4.2 A3-E's Awareness

The *response time* of serverless functions can be decomposed into *network latency* and *service time*. The latter may be further refined into three

parts: *acquisition delay* (Δ_{AQ}), *allocation delay* (Δ_{AL}), and *execution delay* (Δ_E). Most FaaS platforms (e.g. AWS Lambda [4] and OpenWhisk [104]) adopt a *cold start* policy in which containers are allocated after the first call, and kept *warm* for a while even if they are idle [87]. The occasional cold start, however, may be disruptive for latency-sensitive applications.

In light of this, A3-E generalises the *Client Awareness* in the Serverless MEC Architecture (see Chapter 3). A3-E's *Awareness* has two benefits: (i) it enables function acquisition to be also opportunistic (i.e., on actual demand) by triggering the download and deployment of new functions as soon as the client becomes active in a specific domain (e.g., by starting the application or by entering the area of an edge domain), which reduces Δ_{AQ}; and (ii) it alleviates Δ_{AL} by pro-actively warming containers just before they are needed.

The former benefit is crucial for densely distributed edge domains, as it exempts these domains of downloading and installing beforehand all the assets needed for executing all functions in the system. Although storage is a more abundant and less expensive kind of resource, the opportunistic approach favours efficiency by preventing unnecessary usage. This is especially so for less popular or highly localised applications (i.e., the MAR application for tourists). Moreoever, it prevents multiple edge domains fetching all function's assets from a centralised repository at each new admission or update release, and it enhances the operational autonomy of the domain (*discover-and-play*).

The pre-warming of containers refines the practice adopted by some

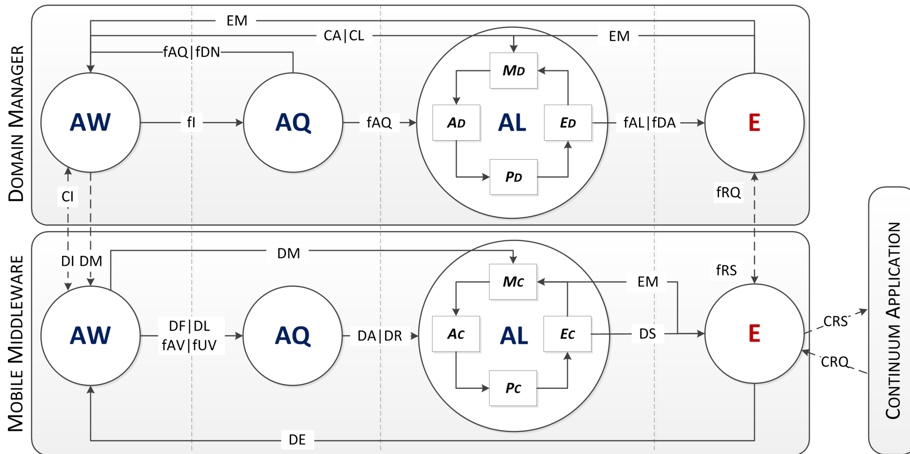


Figure 4.4: *The A3-E framework represented by its constituting activities executed by a Mobile Middleware and a Domain Manager and coordinated through events.*

Table 4.1: *List of events in Figure 4.4*

Source	Sink	Label	Event
Awareness	Awareness	DI	*domain identified*
		CI	*client identified*
		DM	*domain metrics*
Acquisition	Acquisition	fI	*function identified*
		DF	*domain found*
		DL	*domain lost*
	Allocation	CA	*client arrived*
		CL	*client lost*
Acquisition	Awareness	fAQ	*function acquired*
		fDN	*function denied*
	Allocation	fAQ	*function acquired*
		DA	*domain added*
		DR	*domain removed*
Allocation	Engagement	fAL	*function allocated*
		fDA	*function dealocated*
		DS	*domain selection*
Engagement	Awareness	DE	*domain error*
	Allocation	EM	*execution metrics*
	Engagement	fRQ	*function request*
		fRS	*function response*
Client Application	Engagement	C-RQ	*continuum request*
		C-RS	*continuum response*

FaaS platforms. For example, OpenWhisk allows the operator to statically define how many containers are kept pre-warmed for distinct compute runtimes. In contrast with warm containers, a pre-warmed container does not host a function. Still, a considerable fraction of cold start consists of creating the container and not loading the function into the container. As detailed later in this section, Awareness improves the existing approach by enabling container pre-warming to reflect the operating context dynamically.

Edge domains also exploit Awareness by triggering *client join/left* events once mobile devices enter/exit the domain's network. Alternatively, cloud domains rely on special-purpose endpoints (*client join*) and adjustable timeouts succeeding the last request by that client (*client left*). These events allow for a better characterisation of the workload while multiple tenants consume the domain's services. This is particularly important for the efficiency in which resources are allocated, as described in Section 4.4.4.

While cloud domains are not likely to change and may be set up statically through valid Internet names, surrogate edge domains must advertise their existence (*domain identification*) using discovery protocols [66] compatible with their network infrastructure, for example through *Simple Service Discovery Protocol* [134] in local-edge domains; and through *Evolved Multimedia Broadcast/Multicast Service* [53], which are carried over traditional IP multicast over UDP towards end-user devices, in mobile-edge domains. Besides the domain's address within the network, edge domains also need to advertise their API, through which management operations (i.e., client identification, monitoring) and function requests are performed.

From the Mobile Middleware's perspective, *Awareness* is responsible for the discovery of domains (source of *domain found* and *domain lost* events) and for the advertisement of required functions through *client identification* containing their metadata (i.e, `name` and `repository url`). For the duration of the interaction with the domain, Awareness also takes care of keeping track of the availability and QoS metrics (*domain metrics*). The Mobile Middleware uses this information to trigger *function available* and *function unavailable* events whenever the function state in the domain changes (e.g. due to extreme resource contention). More importantly, the middleware harnesses the provided information to feed Allocation with crowd-sourced QoS metrics for each required function.

The communication required for Awareness may be implemented following different approaches, namely:

- Special-purpose, well-known HTTP endpoints with cloud domains.
- Local area network protocols (e.g., IP unicast over UDP) and local HTTP endpoints with local-edge domains.
- Point-to-point radio
- System-level events (e.g., Intent Broadcast in Android) with mobile domains.

4.4.3 A3-E's Acquisition

A3-E's *Acquisition* models the automated download and deployment of the function artefacts from a repository, and the confirmation of the domain's capability of providing a required function(s). Its ultimate goal is to prevent the use of the domain's resources before the function is needed, while also facilitating IT operations (Ops) for developers and administrators.

Ops mitigation is particularly important for densely distributed edge domains since the manual administration of a large number of geographically distributed servers can be cumbersome or infeasible. A self-managed approach towards the discovery of edge domains is also in-line with the concept of *cyber-foraging* introduced by Satyanarayanan [94] —who also states that *the widespread deployment of cloudlet infrastructure (i.e., local-edge domains) will not happen unless software management of that infrastructure is trivial—ideally, it should be totally self-managing [97].* Nevertheless, Ops mitigation can also prove useful for cloud domains: to the best of our knowledge, current FaaS platforms only support uploading (pushing) of functions, often through RESTful APIs [4, 38, 104].

The Domain Manager performs Acquisition by fetching the function artefacts (e.g., compiled classes, dependencies, and static assets) from a repository upon a *client identified* event. Note that mobile domains are exempt from performing Acquisition as local functions are assumed to be downloaded and installed along with continuum applications.

In particular, edge domains may employ different Acquisition approaches. Taking advantage of A3-E's Awareness, densely distributed edge domains will fetch and deploy functions using the following policies:

- Base container images are cached beforehand for all supported compute runtimes.
- Function packages (containing all function assets) are fetched from its uniquely identified repository and deployed to the FaaS platform:
  - upon *function identification* and after the confirmation that the domain is capable of providing the function; or
  - upon admission into the system and following update releases, if the function is specified with pro-active deployment SLA.

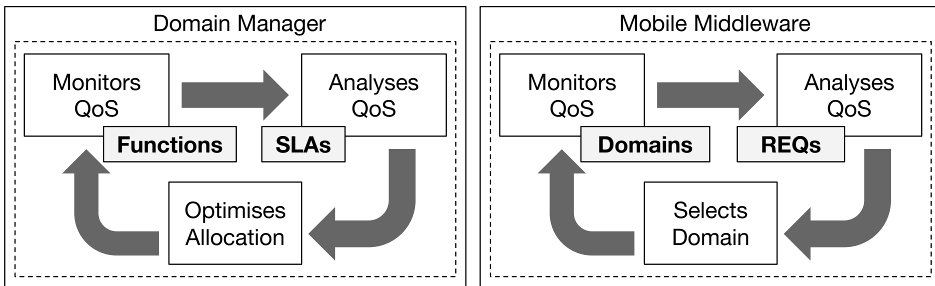
These policies above may be further refined, e.g. using historical usage information and the availability of storage at each node to decide, at runtime, whenever to anticipate the download and deployment of distinct functions and when to deprecate resources from unused functions.

To the Mobile Middleware, *Acquisition* corresponds to the confirmation (or denial) of the capability of a domain in providing the function(s) required by the application. Upon *domain found* event and following *client identification*, the Mobile Middleware awaits for the *domain metrics* notification containing the availability and performance metrics for the functions that are currently available. For each reported function, it then triggers a

function available (or *unavailable*) event —to be handled with the update of the middleware's list of capable domains, and with the subsequent triggering of a *domain added* (or *removed*) event.

In our running example (see Section 4.2.1), the artefacts composing functions from the Image Editing and Mobile Game applications are once fetched and installed by the edge domains in the AV and hotel upon the client identification. Meanwhile, running applications momentarily continue to rely on their mobile domain, or on any other remote domain in which the functions have already been downloaded and deployed due to previous interactions. For example, the mobile-edge domains are likely to skip the acquisition of MAR function artefacts due to previous contacts with other tourists' devices (*function-locality correlation*). Conversely, the AV's and hotel's edge domains are likely to deprecate cached assets from least recently used functions to make space for new ones.

4.4.4 A3-E's Allocation



(a) *Self-management loop performed by the Domain Manager as part of A3-E's Allocation.* **(b)** *Self-management loop performed by the Mobile Middleware as part of A3-E's Allocation.*

Figure 4.5: *Self-management loops for domain-side functions allocation and client-side domain selection*

Domain Manager

The fast and reactive provisioning of containers is the essential characteristic behind the FaaS model efficiency. However, a purely event-driven approach may result in over-provisioning of resources, e.g. in the advent of spikes of workload while the number of warm containers is low (or zero). Employing a minimum queue between function activation and container creation may significantly increase the odds of finding a warm container released following the previous execution. Moreover, the *pre-warming* of

containers is critical for improving responsiveness but needs to account for resource contention among functions and abrupt workload fluctuations.

In light of this, A3-E's Allocation extends the mechanism employed by FaaS platforms with a self-management loop [47]. Its primary goal is to tackle the scaling of warm and pre-warmed containers by remote domains in order to guarantee that the response time of each provided function is in-line with their respective SLA —as formulated in Section 4.3.5. Hence, in contrast with the static pre-warming and purely event-driven spawning of containers, A3-E's Allocation aims to allocate *only the right amount of resources*.

The self-management loop (performed by the abstract MAPE components in Figure 4.4 and further detailed in Figure 4.5a) works as follows:

- The *Monitor* (M_D) keeps track of function *execution metrics* (EM) in terms of response time; it also detects *client arrived/left*, which are used to further characterise the workload in terms of active users.

- The *Analyser* (A_D) then aggregates these data over a predefined time window and computes the *arrival rate* (α_j), *average response time* (τ_j), and the *number of clients* (c_j) for each function.

- The *Planner* (P_D) considers the analysis results and calculates the overall number of pre-warmed ($\bar{p}$) and warmed ($\bar{c}$) containers, so that $\tau_j \leq \Delta_j \quad s.t. \quad c_j \leq Cmax_j, \; \forall f_j \in F$, as defined in Section 4.3.5.

- Closing the loop, the *Executor* (E_D) carries out the new allocation scheme with the creation and termination of containers so that the number of pre-warmed and warm containers per function matches the planned allocation.

Given the limitations of densely distributed edge domains, resource contention between different functions may exist. The planner is also in charge of managing such situations, which may be achieved through distinct policies and heuristics. In our Running Example, critical applications (e.g. Autonomous Vehicles) should have higher priority over less critical ones (e.g. the MAR application for tourists). Another possibility is to distinguish assurances by their revenue to the edge domain operator (e.g.. gold vs silver SLA). In such cases, some functions (e.g. less critical) might become unavailable or available with a slower response time at the edge domain, prompting the choice of alternative domains (e.g., mobile, cloud, or edge domains from different providers).

Mobile Middleware

The Mobile Middleware handles QoS fluctuations through the dynamic selection of the domain best satisfying function requirements. Analogously to the domain-side Allocation, the Mobile Middleware realises this activity using a self-management loop [47] for each function required by the applications it is currently managing. At each iteration, the self-management loop (performed by the abstract MAPE components in Figure 4.4 and further detailed in Figure 4.5b) works as follows:

- Through Awareness, the *Monitor* (M_C) collects crowd-sourced performance metrics (i.e., *service time*) and the *network latency* for all capable domains (set D in Section 4.3.5); it also collects *execution metrics* from Engagement to keep track of the updated *battery consumption* and *response time* for the selected domain.
- For a predefined time window, the *Analyser* (A_C) aggregates monitored data to compute the overall *battery consumption* and *response time* of each capable domain.
- The *Planner* (P_C) uses the analysed data to run a multi-criteria rating algorithm [86] and compute the best domain for the next invocations.
- Closing the self-management loop, the *Executor* (E_C) enacts the selection (if the computed domain differs from the one in use) by triggering a *domain selected* event.

Meanwhile, the Domain Manager's Planner handles *domain removed* events (e.g. following network disconnection or function unavailability) by disregarding that domain in subsequent iterations and removing it from the current selection list. This condition is reverted upon *domain added* event.

4.4.5 A3-E's Engagement

A3-E's *Engagement* models the actual provisioning of a serverless function by a domain after successful Acquisition and Allocation. Throughout Engagement, and as long as the mobile-domain interaction persists, Continuum Applications can engage with that selected domain by triggering function invocations.

Remote domains (i.e., cloud and edge) are engaged through distributed protocols (e.g., HTTP requests and WebSockets). To enforce a standard interface between the mobile and remote domains, the former also expose its functionality as local services.

During Engagement, a remote Domain Manager forwards *function request* events to the FaaS platform, which in turn maps each request to a containerised compute runtime responsible for its execution (see Section 2.4). The decisions taken by A3-E's Allocation are enacted as follows:

- Upon *function deallocated* event, the Domain Manager scales down to zero the number of containers for that function, which causes subsequent invocations to be queued.
- Upon timeout, the domain responds (*function response*) with a specific error code (e.g. service unavailable) for all queued requests.
- Conversely, a *function allocated* event indicates the recovery of a given function; the number of warm containers is restored, and the FaaS platform resumes the processing of requests.

From the Mobile Middleware's viewpoint, a *C-request* event indicates that a continuum application has sent a new request to a serverless function. C-requests contains the name of the target function along with required parameters; the middleware handles it by performing a *function request* addressing the currently selected domain. Upon *function response* arrival, the middleware triggers a *C-response* event handled by the Continuum Application.

During *Engagement*, the Mobile Middleware continuously listens for *domain selected* events containing an updated rank of best domains. The middleware harnesses the availability of multiple domains to provide fault tolerance; whenever a domain fails (e.g. service unavailable), the next best domain is used. If none is available for that specific function, the middleware reacts by queuing subsequent *C-requests* until a *domain selected* event confirms a new domain or, upon timeout, it replies the Continuum Application with a *C-response* containing the corresponding error.

4.4.6 Running Example

Figure 4.6 depicts a timeline of events from our Running Example. The timeline starts with our user initialising the MAR application after entering the touristic area. The Mobile Middleware receives a *domain identification* event (DI) from its mobile domain, and replies with a *client identification* (CI). Following the function identification (fI), the middleware triggers a *function allocated* (fAL) once the required functions have been registered. After selecting this domain, the middleware triggers a *domain selection* (DS), which allows subsequent *C-request* (CRQ) events to be handled locally.

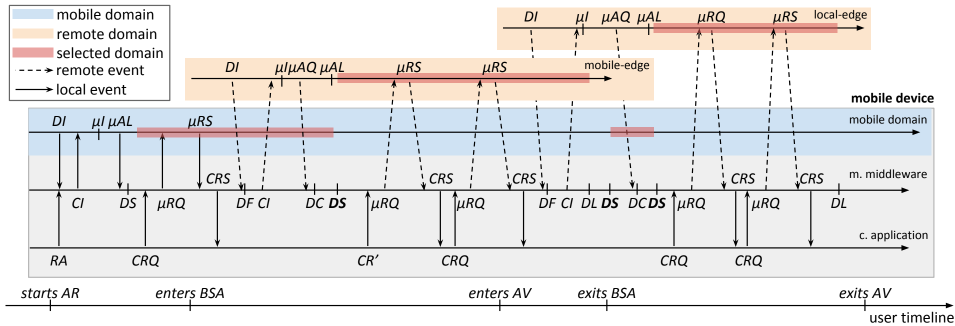


Figure 4.6: *A timeline of events from the Running Example scenario.*

As our user enters the area covered by a mobile-edge domain serving a base station area (BSA), the middleware receives a *domain identification* (DI) and repeats the previous set-up procedure. To prevent battery drain, the Mobile Middleware decides for the mobile-edge domain and triggers a new DS event once the *function acquired* (fAQ) event arrives. After a long period of engagement with the mobile-edge domain, our user enters the AV and its local-edge domain. Due to a change of network, the connection with the mobile-edge domain is lost (DL). To prevent service interruption, the middleware momentarily switches back to the mobile domain (DS). Meanwhile, as this is the first contact with the MAR application, the edge-domain goes through *Acquisition*, which takes some time (ΔAQ) to complete (fAQ). Upon a *domain confirmed* event (DC), the AV's local-edge domain is selected (DS) and engaged throughout the journey.

4.5 Domain Manager

To demonstrate the feasibility of A3-E in managing the life-cycle of continuum functions, we implemented a prototype version of the *Domain Manager*. While MEC architecture is certainly interesting, it is also significantly more difficult to access to cellular infrastructure and technology. Thus, we focus on the implementation of a local-edge domain co-located with our department's infrastructure and accessed through WiFi, similarly to cloudlets [97].

Notwithstanding the differences regarding communication technology (i.e., LTE broadband) and supporting components (i.e., responsible for authenticating user requests and managing traffic rules) between local- and mobile-edge domains, we argue that the central concepts of the proposal —namely, the opportunistic deployment of serverless functions— would

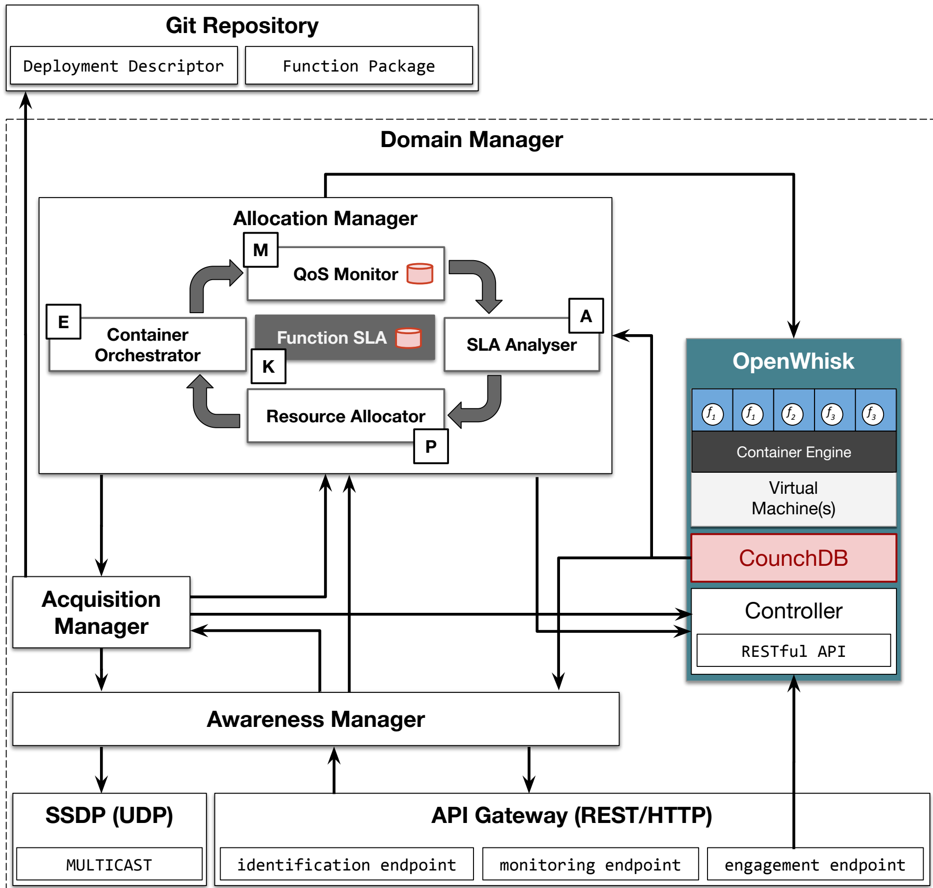


Figure 4.7: *Domain Manager architecture overview.*

be shared with mobile-edge domains extending the Serverless MEC Architecture presented in Chapter 3. We also argue, for the scope of the contributions in this work, edge domains are significantly more complex than cloud domains, which may still benefit from many of the techniques and mechanisms presented in this section.

Note that, for the sake of presentation, we opt for describing the implementation of the *Mobile Middleware* and accompanying *mobile domain* in a separated section (Section 4.6). As reported later in this work, these prototypes were employed in the experimental evaluation of the Mobile-Edge-Cloud Continuum and the A3-E framework.

4.5.1 Architecture Overview

The Domain Manager was implemented[1] using a modular architecture reflecting A3-E's activities. More precisely, *Awareness*, *Acquisition*, and *Allocation* were implemented as separated modules in Python language. While scalability might be a concern, Python language has an active community and features several libraries that reduce the burden of implementing the prototype. It therefore offers good cost-benefit to demonstrate the Domain Manager's functionality.

Our prototype relies upon *OpenWhisk* [104], the most mature open source solution for handling the execution of serverless functions. Our solution interfaces with OpenWhisk through its (management) RESTful API. The prototype also harnesses the functionality of some of the components in OpenWhisk technology stack (e.g. *CouchDB NoSQL Database* and *Ngix*).

Figure 4.7 gives an overview of the Domain Manager architecture. Note that, throughout the remainder of this section, we shall refer to A3-E's activities and their respective *Manager* component interchangeably.

4.5.2 Awareness Manager

The Awareness Manager handles part of the mutual awareness between mobile devices and the domain. Local-edge domains are co-located with wireless local area network infrastructure. We leverage the *Simple Service Discovery Protocol* (SSDP) [134], a state-of-the-art network protocol, for implementing the *domain identification* procedure in A3-E's Awareness.

The SSDP protocol is designed to work without the assistance of server-based configuration mechanisms, such as *Dynamic Host Configuration Protocol* (DHCP) or *Domain Name System* (DNS), and without special static configuration of a network host. SSDP uses UDP to multicast text-based `NOTIFY` messages to a well-known IP address (`239.255.255.250`) and port (`1900`). In addition to multicast (*passive discovery*), interested devices may also query (*active discovery*) using the M-SEARCH method. Responses are sent via unicast to the originating address and port number of the multicast request.

The *Awareness Manager* utilises the SSDP protocol to advertise its presence and interface within a *network domain*. The Mobile Middleware (presented later in Section 4.6) uses the same protocol to listen to `NOTIFY` messages and acquire information about the domain's RESTful API, namely:

- **identification endpoint**: used for client identification in Awareness.

[1] Documentation and source code available at `https://github.com/deib-polimi/A3-E-DSM-local-edge/`

- **monitoring endpoint**: used by Awareness to keep track of the domain status (i.e., function availability) and crowd-sourced performance metrics (i.e., the average response time of available functions).
- **activation endpoint**: used to trigger the activation of functions during Engagement.

In this set-up, we follow the assumption that an *edge domain* will overlap with the *network domain* in which it advertises its presence through the UDP-based SSDP protocol. This is in-line with the domain abstraction rationale presented in Section 4.2. Consequently, a single *Domain Manager* is expected to handle the control of the FaaS platform —deployed to one or multiple (virtual) machines— serving the entire network domain.

Mobile devices within a domain network are not always engaging with that domain. The Awareness Manager provides the critical functionality of exposing, through the *monitoring endpoint*, updated information regarding the availability and performance of admitted functions, as requested by the Mobile Middleware. The rationale, inspired by Flores et al. [30], is that mobile devices can take advantage of crowd-sourced execution metrics. For each requested function, the Domain Manager checks through OpenWhisk APIs if it is currently available. Additionally, it queries *CouchDB* (the NoSQL composing OpenWhisk's architecture) to obtain the aggregate service time from past function activations.

Listing 4.1 shows an example of a *domain metrics* message containing the availability status and crowd-sourced service time for three functions.

4.5.3 Acquisition Manager

The Acquisition Manager's main thread subscribes to Awareness events. For each required function contained in *client identification*, the manager proceeds concurrently with A3-E's *Acquisition*. First, it checks whether the function's credentials (i.e., name and repository URL) are present in a *whitelist*. This simple positive control allows the edge domain administrator to restrict access and prevent misuse of the domain's services by unauthorised applications. Edge domains targeting unspecified applications (e.g., hotel guests in our Running Example) may opt for granting access to the domain's services based on network-level or user-level authentication.

Amongst the files in the function's repository, a *Deployment Descriptor* provides deployment instructions. Listing 4.2 provides an example of a function Deployment Descriptor file. The descriptor is written using *JavaScript Object Notation* (JSON), an open-standard key-value file

Listing 4.1: *Example of deployment descriptor file used by remote domains.*

```
{
  "metrics": [
      {
        "repo": "https://github.com/ste23droid/A3E-face-
           detection",
        "status": "available",
        "serviceTime":
            {
                "value": 0.11072,
                "unit": "ms"
            }
      },
      {
        "repo": "https://github.com/ste23droid/A3E-POI-
           recognition",
        "status": "available",
        "serviceTime":
            {
                "value": 0.19034,
                "unit": "ms"
            }
      },
      {
        "repo": "https://github.com/ste23droid/A3E-neural-
           transfer",
        "status": "unavailable"
      }
  ]
}
```

format. At the current version, function deployment descriptor files are expected to contain the following attributes:

- **functionName:** unique `name` within the user namespace; passed "as it is" to the OpenWhisk CLI.
- **path:** relative `path` to the compressed package (`.zip`) containing the function's assets; passed "as it is" to OpenWhisk CLI.
- **runtime**: the *compute runtime*, accordingly to the domain's catalogue.
- **dependencies**: list of software dependencies required to run the function; each dependency is specified by its `name` and `version`.

- **memory:** how much memory is assigned to the OpenWhisk action.
- **timeout:** the maximum execution time before a timeout error.

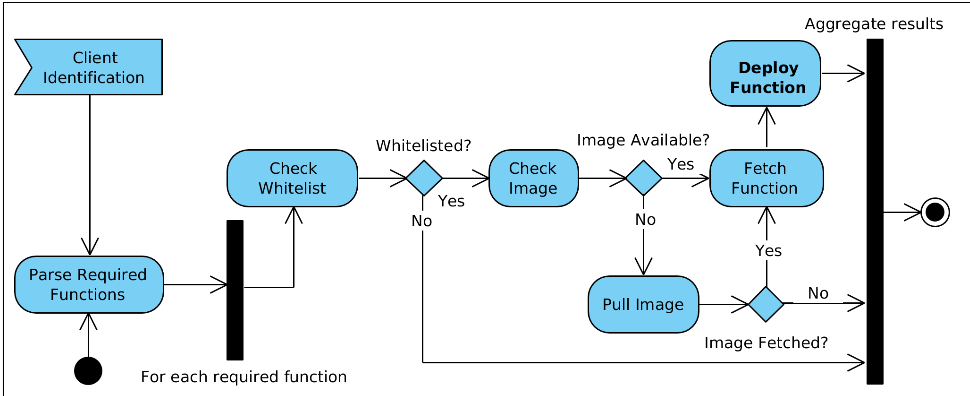


Figure 4.8: *Activity diagram depicting the function identification procedure.*

A domain is considered capable if: (i) the requested compute runtime is registered in the domain's catalogue and the corresponding Docker container image is available; (ii) all software dependencies are satisfied; and (iii) required storage is available.

In theory, container images may be retrieved at runtime from public repositories (e.g. *Docker Hub*). For instance, one can find bundles of lean operating system containers (e.g. *Alpine Linux*) plus runtime (e.g. *Python*) as small as `60MB`. In practice, a function often relies on other dependencies; the total size of the container image can jump to hundreds of megabytes or more. Hence, the manager conditions the domain's capability in serving a function to the local availability of the corresponding container image. If the function is whitelisted, the Acquisition Manager will attempt to pull a missing image. Otherwise, the domain is denied for that function. It is only after verifying and confirming the domain's capabilities in providing the required functionality that the Acquisition Manager will fetch from the same repository the compressed package containing all function assets.

The activity diagram in Figure 4.8 depicts the Acquisition procedure. Upon a client identification event (i.e., an HTTP request), the manager parses the set of required functions. For each function (depicted by the fork bar in the diagram) the manager will check for the function's name and repository in the whitelist. If it is not available, it will then pull the image before finally fetching the function package.

Once the function package and dependencies are ready, the manager proceeds with its deployment using OpenWhisk's command-line interface. The different threads handling individual acquisition will eventually join (depicted by the join bar in the diagram). The outcome of each operation is sent as an aggregate response to the Mobile Middleware. If successful, each operation triggers an individual *function acquired* event, which is handled by the domain's *Allocation Manager*.

Listing 4.2: *Example of Deployment Descriptor file used by remote domains.*

```
{
    "functionName": "faceDetection",
    "path": "faceDetection.zip",
    "runtime": "Python",
    "dependencies": [
      {
        "name": "numpy",
        "version": ">=1.15"
      },
      {
       "name": "opencv",
       "version": ">=3.4.2"
      }
    ],
    "memory": 256,
    "timeout": 2
}
```

4.5.4 Allocation Manager

The Allocation Manager is responsible for guiding the FaaS platform's allocation process, according to the domain's context of operation and the SLA of deployed functions. We propose a fast and scalable materialisation of the self-management loop in A3-E's Allocation based on control theory to cope with the requirement of continuum applications and highly dynamic workload (e.g., from users that quickly enter and leave a particular area).

In literature, one can find many approaches for dynamic resource allocation based on heuristics [99], artificial intelligence [74] and queue theory [40]. We adopted an extremely lightweight control theoretic technique that exploits the container technology [10].

The primary goal of the control-theoretic allocation is to prevent the

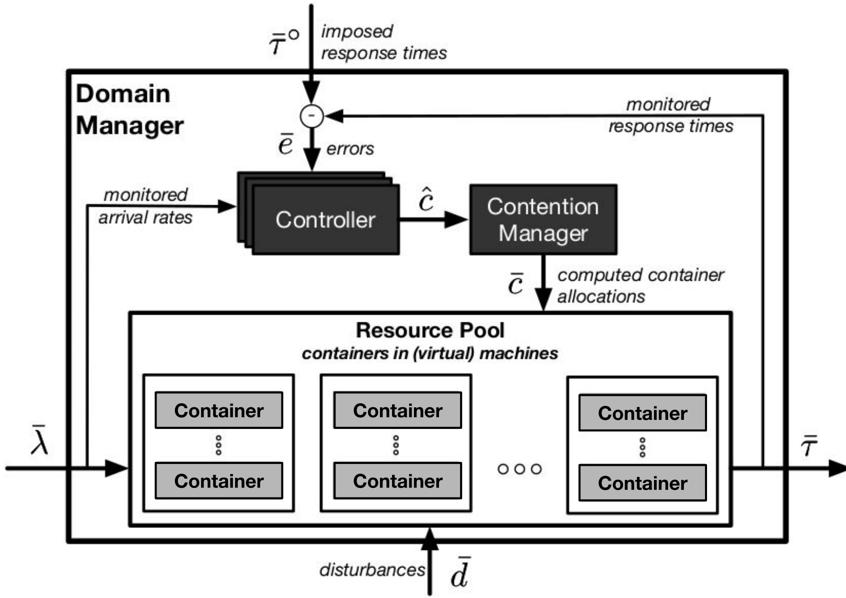


Figure 4.9: *Control loop implementing the self-management in A3-E's Allocation.*

overallocation of containers in the advent of workload spikes. While cloud-based FaaS operators enjoy virtually unlimited resources, overallocation threatens the performance of contender functions in edge domains due to limited horizontal scalability. Simultaneously, controllers can readily change resource allocation (i.e., short control period), thanks to the reduced time needed to create and load functions into containers.

Control-Theoretic Allocation

Figure 4.9 gives an overview of the control system, which is responsible for limiting the number of containers to the cluster of (virtual) machines composing a domain's pool of resources (or the *plant* in control-theoretical terms). Each container provides the runtime environment required for the execution of a given continuum function; and multiple containers may co-exist in one or more machines (horizontal scalability).

The control plant is subject to different signals that could be measurable (input variables) or unknown (disturbances). Considering a discrete time, for each admitted serverless function we define $\lambda(k)$ as the function of the measured arrival rate of requests at each control time k; and $\bar{\lambda}(k)$ as the corresponding vector for all admitted functions.

At time k, the function is executed in a $c_j(k)$ number of allocated con-

tainers, while $\bar{c}(k)$ is the corresponding vector for all functions. The disturbances are defined as $\bar{d}$ and, by definition, cannot be directly controlled and measured. Finally, $\bar{\tau}$ is the system output and corresponds to the response time vector comprising all functions, whereas $\bar{\tau}^\circ$ corresponds to the vector of desired response time per function (or control *set-point*) according to the SLA of each function.

In our current set-up, the function $\bar{\tau}^\circ(k)$ does not vary over time, meaning a constant targeted response time for each function. Of course, these values should be set below the SLA threshold (i.e., $\tau_j^\circ(k) \leq \Delta_j, \forall\ k \in \mathbb{N}, f_j \in F$). Moreover, since response time cannot be measured instantaneously but by aggregating the execution time of multiple requests over a predefined time window, many aggregation techniques could be used without any change to the system model and controller. In our framework, we compute the average of the response time values in $\bar{\tau}$ within each control period, but stricter aggregation functions like the $99th$ percentile could be used depending on the needs of the service provider.

We dedicate one *Controller* per function with the goal to compute the next container allocation in order to obtain:

$$\tau_j(k)^\circ - \tau_j(k) = e_j(k) = 0 \quad \forall k \geq 0 \tag{4.1}$$

or more realistically, considering all functions:

$$\bar{e}_j(k) \simeq 0 \quad \forall k \geq 0, \forall f_j \in F \tag{4.2}$$

To achieve this objective controllers must model the system by defining a characteristic function. We assume that this function does not need to be linear but regular enough to be linearisable in the domain space of interest. Moreover, we consider this function to be dependent on the ratio of the number of allocated containers c and the request rate λ.

The characteristic function (f to give it a name) is intuitively monotonically decreasing towards a possible lower horizontal asymptote, as it can be assumed that once the parallelism degree of a function is fulfilled by the available containers, adding new ones causes no further decrease in the response time. More specifically, we found a practically acceptable function to be:

$$f\left(\frac{c(k)}{\lambda(k)}\right) = \widetilde{u}(k) = c_1 + \frac{c_2}{1 + c_3\frac{c(k)}{\lambda(k)}} \tag{4.3}$$

where parameters c_1, c_2, and c_3 were obtained through profiling of each function. Thus we obtain the following dynamic system:

$$\tau(k) = p\tau(k-1) + (1-p)\widetilde{u}(k-1) \tag{4.4}$$

where $p \in [0,1)$ is the single pole of the system estimated with step response analysis. As control technique, we rely on PI controllers because they are able to effectively control systems dominated by a first order dynamic [6] (i.e., representable with first order differential equations) such as the studied ones. PI controllers compute the next state of a plant by using two contributions: one that is proportional and another that is integral to the error e. Algorithmically, $\forall\ k \in \mathbb{N}, f_j \in F$:

$$\begin{aligned}
e &:= \tau_r^\circ - \tau_r;\\
x_R &:= x_{R_p} + (1-p) * e_p;\\
c &:= \lambda * f_{inv}((\alpha - 1)/(p-1) * (x_R + e));\\
c &:= max(min(Kmax, c), Kmin);\\
x_{R_p} &:= (p-1)/(\alpha - 1) * f(c/\lambda) - e;\\
e_p &:= e;
\end{aligned}$$

where the "p" subscript denotes "previous" values (i.e., those corresponding to the previous step) and "f_{inv}" corresponds to the inverse of the characteristic function, and x_R the state of the controller. Finally, $\alpha \in [0,1)$ and $p \in [0,1)$ are the single pole of the controller and the system, respectively. The higher the value of α, the faster will be the error convergence —ideally to zero— at the expense of a more fluctuating allocation.

Resource Contention

At each control step, function controllers run independently (i.e. without synchronization) to compute the next container scale for the corresponding function. Vector $\hat{c}$ contains the number of containers for of all functions. Nonetheless, $\hat{c}$ can not immediately actuated since the sum of containers could exceed the domain's capabilities (e.g. due to resource contention among functions).

We tackle the scenario above with two policies: *priority first*, and *fairness*. More precisely, $\hat{c}$ is passed to another control component, namely the *ContentionManager*. The latter outputs a vector $\bar{c}$ corresponding to the actual allocation; $\bar{c}$ is defined as follows:

$$\bar{c}(k) = \begin{cases} \hat{c}(k), & \text{if } \textit{no resource contention} \\ solveContention(\hat{c}(k)), & \text{otherwise} \end{cases} \tag{4.5}$$

where function `solveContention` scales down the container allocation in $\bar{c}$ to be equal to the maximum memory capacity. In case of resource contention among distinct functions, allocation demand is distributed first among functions of higher priority (*priority first*) until their demand is fulfilled or capacity is exhausted. Functions with the same priority are given an equal number of containers (*fairness*). The *ContentionManager* also updates the state of each Controller (variable x_{R_p}) to reflect the actual container allocation and thereby assure Controllers' consistency.

4.5.5 Engagement

The Domain Manager's prototype delegates A3-E's Engagement to OpenWhisk. We also take advantage of the components natively employed by OpenWhisk —namely, Nginx as the API Gateway— to proxy external *function requests* to the FaaS Platform. To this end, the *engagement endpoint* advertised by the Awareness Manager refers to the actual API exposed by the FaaS platform's Gateway component.

For each arriving function request, OpenWhisk's *Controller* identifies the activated function and dispatches its execution. Both activation and outcome are persisted to the CouchDB NoSQL database. These records become the source of the crowd-sourced performance metrics gathered and reported by the Awareness Manager upon request from the Mobile Middleware's fault tolerance mechanism.

The current prototype does not queue requests upon *function deallocation* events from the Allocation Manager. Nonetheless, the platform replies with an error code indicating that the function is not currently available, which is then handled by the Mobile Middleware.

4.6 Mobile Middleware

In this section, we describe the implementation[2] of the Mobile Middleware and accompanying the mobile domain. In particular, our implementation targets the Android platform, which in 2018 achieved almost 75% of the smartphone market share.

4.6.1 Architecture Overview

The Mobile Middleware architecture follows the guidelines for the design of autonomic elements [47, 81], i.e., it comprises an *Autonomic Manager*

[2] Documentation and source code available at `https://github.com/deib-polimi/A3-E-CSM`

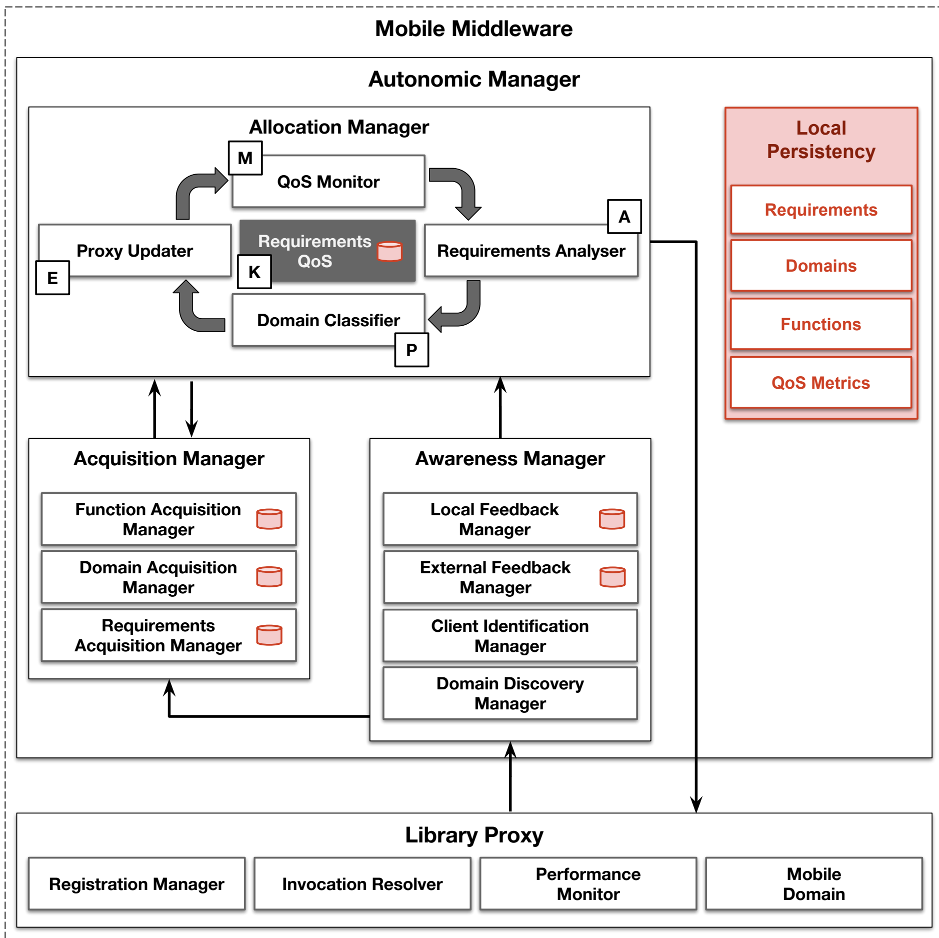


Figure 4.10: *Mobile Middleware architecture overview.*

—responsible for self-managing of a functional unit— and the managed component —in our case, the *Library Proxy* component.

Similarly to the Domain Manager, the Autonomic Manager is implemented as separated *Manager* components corresponding to its three first A3-E activities: the *Awareness Manager*, the *Acquisition Manager*, and the *Allocation Manager*. Each of these is refined into sub-components that take care of specific concerns in a modular fashion.

As the name suggests, the *Library Proxy* consists of an Android library that intermediates the interaction between client applications hosted by the mobile device and various domains. As a *managed component*, it provides the Autonomic Manager with monitoring information —namely, local per-

formance metrics— and it receives directives for its behaviour —namely, the ranked list of domains for each function required by hosted applications.

4.6.2 Continuum Application Registration

Continuum applications register themselves to the Mobile Middleware passing an application configuration (*App Config*) file. Similarly to the function Deployment Descriptor file in Section 4.5.3, the App Config file is formatted in JSON. Besides the Continuum Application's unique identifier (`appId`), the file contains the metadata and requirements for all functions consumed by this applications, namely:

- **name:** the unique name that uniquely identifies the function within the Continuum Application namespace.
- **repo:** the repository URL containing the function Deployment Descriptor and package files used to deploy the serverless function on remote domains (see Section 4.5.3 for details).
- **qosRequirements:** set of QoS requirements for this function (see Section 4.3.2), namely:
  - **responseTime**
    - **weight:** default response time weight ($\in [0, 2]$).
    - **threshold:** default response time threshold ($\in [0, 2]$).
  - **batteryConsumption**
    - **weight:** default battery consumption weight ($\in [0, 2]$).
    - **threshold:** default battery consumption threshold ($\in [0, 2]$).
- **locationRequirements:** set of *Location-Requirements* for this function (see Section 4.3.2), namely:
  - **mobileDomain**
    - **enabled:** `true` if the Mobile Domain provides a local implementation of this function.
    - **path:** fully qualified path name of the file implementing this function within the application's classpath.
  - **edgeDomains**
    - **enabled:** `true` if the developer wants to enable the support of edge domains for this function.

    * **type:** `ANY|LOCAL|MOBILE`

  - **cloudDomains**

    * **enabled:** `true` if the developer wants to enable the support of cloud domains for this function.
    * **endpoint:** fully qualified URL to be used by the Library Proxy to invoke this function on the cloud domain.

The motivation for using a configuration file instead of class level *annotations* is twofold. First, it allows remote-only functions to be registered without the need for creating a local empty function. Secondly, it enforces a portable format that may be shared with other mobile platforms. Listing 4.3 provides an example of an App Config file. Additional details on the use of each attribute by the prototype are provided later in this section.

4.6.3 Awareness Manager

The *Awareness Manager* fulfils the other part of the mutual awareness contract between mobile devices and Continuum domains.

Its first role regards the discovery of surrogate edge domains. The *Domain Discovery Manager* listens for UDP messages multicasted by edge domains using the SSDP protocol (*domain identification*). To avoid battery drain, the discovery manager limits its activity to a short period after the Mobile Middleware is launched or the network connection changes (e.g. from 5G network to a local-area wireless network). As described in Section 4.5.2, the SSDP `NOTIFY` message contains the following information: the *identification endpoint*, used to send *client identification* requests; the *engagement endpoint*, based on which function requests endpoints are composed; and the *monitoring endpoint*, used by the *External Feedback Manager* to obtain crowd-sourced performance metrics.

The Domain Discovery Manager leverages native networking utility offered by the Android SDK (viz, `isReachable`) to keep updated knowledge about the network reachability and latency of remote domains. Similarly to the crowd-sourced performance metrics discussed below, this procedure provides the Mobile Middleware with awareness regarding remote domains that are not currently engaged. Moreover, this procedure is the source of both *domain found* and *domain lost* events.

For every *domain found* event, the *Function Acquisition Manager* proceeds by sending a POST request to the domain's *identification endpoint* containing the address of the repository of all registered functions at that

Listing 4.3: *Example of App Config file.*

```
{
  "appId": "com.theapp.facedetection",
  "functions": [
    {
      "name": "faceDetection",
      "repo": "https://github.com/ste23droid/A3E-face-detection",
      "qosRequirements": [
        {
          "metric": "responseTime",
          "weight": 2,
          "responsivenessThreshold": 2
        },
        {
          "metric": "batteryConsumption",
          "weight": 2,
          "threshold": 2
        }
      ],
      "locationRequirements":
      {
        "mobileDomain": {
          "enabled": true,
          "path": "app/continuum/functions/faceDetection.js"
        },
        "edgeDomains": [
          {
            "enabled": true,
            "type": "ANY"
          }
        ],
        "cloudDomains": [
          {
            "enabled": true,
            "endpoint": "http://faasprovider.com/api/identification"
          }
        ],
      },
    }
}
```

Listing 4.4: *Example of client identification response.*


```
{
  "functions": [
      {
        "repo": "https://github.com/ste23droid/A3E-face-
           detection",
        "compatible": true,
        "name": "ste23droid/faceDetection"
      },
      {
        "repo": "https://github.com/ste23droid/A3E-neural-
           transfer",
        "compatible": false
      }
  ]
}
```


moment (see Section 4.5.3). The client identification response contains information about which functions are compatible/incompatible; for the former, the name of their URI within the FaaS platform's namespace.

Listing 4.4 shows an example of a client identification response. Each function is uniquely identified by its repository url. For each confirmed (denied) function, a system level *function available (unavailable)* event is triggered —to be handled by the *Acquisition Manager*.

At regular intervals and upon new Continuum Application registration, the *External Feedback Manager* queries the domain through its *monitoring endpoint*. Not only it keeps track of the availability of required functions, but it also collects crowd-sourced performance metrics.

The availability of functions is mainly affected by the decisions taken by the domain's Allocation Manager in the advent of resource contention (see Section 4.5.4). In contrast, crowd-sourced performance metrics are aggregate from previous executions from multiple clients. This approach allows the Mobile Middleware to oversee the domains that are not currently been engaged, i.e., domains from which performance metrics are not collected locally. Listing 4.5 provides an example of a domain report (in JSON format) containing the status and performance metrics of three functions.

Additionally to the availability and service time, the internal *domain metrics* event triggered by the External Feedback Manager contains the network latency most recently acquired by the Domain Discovery Manager. While we could have opted to use the response time to the domain metrics

Listing 4.5: *Example of domain metrics response.*


```
{
  "metrics": [
      {
        "repo": "https://github.com/ste23droid/A3E-face-
           detection",
        "status": "available",
        "serviceTime":
            {
                "value": 0.11072,
                "unit": "ms"
            }
      },
      {
        "repo": "https://github.com/ste23droid/A3E-POI-
           recognition",
        "status": "available",
        "serviceTime":
            {
                "value": 0.19034,
                "unit": "ms"
            }
      },
      {
        "repo": "https://github.com/ste23droid/A3E-neural-
           transfer",
        "status": "unavailable"
      }
  ]
}
```


HTTP request to obtain an estimated network latency, the latter would also include the time needed to process such request. In contrast, the network latency reported by the Domain Discovery Manager is free of such biases.

Last but not least, the *Local Feedback Manager* performs the important role of retrieving from the Library Proxy the performance metrics from local and remote execution of functions by the selected domain(s): after registration, continuum applications are able to send requests to the Continuum through the Library Proxy API. Each request is ultimately dispatched for execution by the proxy based on the ranked list of best domains; raw execution metrics —both local and remote— are then batched in memory and periodically sent to the Local Feedback Manager for analysis.

Listing 4.6: *Example of local execution metrics notification.*

```
{
  "package": "com.example.app",
  "localMetrics": [
    {
      "functionName": "faceDetection",
      "payloadBytes": 65000,
      "responseTime": 0.383,
      "ticks": 284.37
    },
    {
      "functionName": "faceDetection",
      "payloadBytes": 81250,
      "responseTime": 0.402,
      "ticks": 289.55
    }
  ]
}
{
  "package": "com.example.app",
  "localMetrics": [
      {
        "functionName": "faceDetection",
        "payloadBytes": 65000,
        "responseTime": 0.383,
        "ticksPerSec": 284.37
      },
      {
        "functionName": "faceDetection",
        "payloadBytes": 81250,
        "responseTime": 0.402,
        "ticksPerSec": 289.55
      }
  ]
}
```

Listing 4.6 shows an example of the execution metrics (in JSON format) reported by the Library Proxy.

4.6.4 Acquisition Manager

The Acquisition Manager is responsible for keeping updated the Local Persistence registries with the functions required by continuum applications

(Requirements); the availability of domains (Domains); and the catalogue of functions provided by each domain (Functions). In particular, the use of a local persistence component prevents the loss of state when the Mobile Middleware is terminated by the operating system —a common routine with Android platforms in case of resource contention or inactivity.

The *Requirements Acquisition Manager* acquires information about the requirements from continuum applications currently active in the mobile device. More precisely, applications leverage the Library Proxy API to register themselves passing the App Config file through the `registerApp` interface. In turn, the manager takes care of parsing the App Config file and registering the required functions.

The *Function Acquisition Manager* and the *Domain Acquisition Manager* react to internal *function available/unavailable* and *domain found/lost* events by adding/removing the corresponding entity to/from the Local Persistence component. These components will also react to each event by triggering a corresponding *domain added/removed* event.

4.6.5 Allocation Manager

The Allocation Manager implements the self-managing loop introduced in Section 4.4.4 and performed for each of the functions in the registry, taking into account their requirements and the QoS of each domain.

Monitoring and Analysis

The loop starts with the *QoS Monitor*. At each cycle, this component keeps track of the most recent metrics regarding the QoS of different domains. To this end, it listens to two sorts of events: *domain metrics*, reported by the External Feedback Manager; and *execution metrics*, reported by the Local Feedback Manager.

The raw metrics collected are then passed to the next component, viz. the *Requirements Analyser*. The role of this component is to aggregate the QoS metrics from the previous step and produce useful information regarding the QoS-Requirements specified by the application developer. The aggregate metrics are *response time*, which comprises both service time and network latency; and *battery consumption*, which is either a result of the local function execution or the networking with remote domains.

In the literature, one can find multiple energy modelling techniques. Nonetheless, the granularity level characteristic of serverless functions makes it more difficult to obtain precise information regarding the battery drain inflicted by functions in isolation. Hence, many of the existing approaches

are not suitable for this challenge.

More recently, a method-level energy model was proposed by Neto as part of the ULOOF framework [73]. The proposed energy model considers both the CPU and various network interfaces, which makes it suitable for A3-E and the Mobile-Edge-Cloud Continuum. In their original model, the energy consumption inflicted by method m while using the CPU is defined as:

$$CpuConsumption(m) = l_{cpu}(cpuTicks(m)/execTime(m)) * execTime(m)$$

For the same method, the energy consumption while using the network interface i is defined as:

$$NetworkConsumption(m,i) = l_{radio,i}(throughput(i)) * transfTime(m)$$

Where l_{cpu} and $l_{radio}(i)$ functions are device specific and profiled offline using a USB Power Meter [73]. Next, we briefly describe how each of the additional variables in the previous model can be obtained in the context of Android platform devices and A3-E.

Cpu Ticks. In an operating system, a tick consists of an arbitrary unit for measuring internal system time from which various operating system functions are derived. The *CPU tick frequency* is OS dependent and may be obtained through kernel level constants. The Android operating system is based on the Linux kernel; the CPU tick frequency can obtain through the `_SC_CLK_TCK` constant and equals 100, i.e., one tick every $10ms$. This information allows us to calculate the time spent by the CPU on a block of code using the following equation:

$$execTime = \frac{utine + stime}{_SC_CLK_TCK}$$

Where `utime` and `stime` are the CPU ticks respectively spent in the user and kernel code. For a given process (`PID`) and thread (`TID`), both values can be obtained through the `/proc/<PID>/task/tid/stat` system file.

Throughput. The throughput of a network interface will vary according to the technology. Moreover, wireless interfaces are subject to noise; according to the Shannon-Hartley's theorem:

$$C = B\log_2(1 + SNR)$$

where C is the channel capacity in bits per second, B is the channel bandwidth in hertz, and SNR is the signal-to-noise ratio.

In Android, the current WiFi throughput was obtained programmatically through the networking utility provided by its SDK (`getLinkSpeed`). Alternatively, the throughput of a mobile data link is calculated using the Shannon-Hartley's theorem based on the nominal link capacity of each broadband technology (e.g. $7.2Mbps$ in HSDPA, $42Mbps$ in HSPA+, and $150Mbps$ in LTE) and the current SNR value, which can also be obtained programmatically.

Once the throughput for the network interface associated with a given domain is determined, we may use it to estimate the time needed to transfer the function payload and receive its execution output through the equation:

$$transfTime = \frac{inputSize + outputSize}{throughput}$$

Of course, in most cases, it can not determine the precise function output size. Nonetheless, this uncertainty can be mitigated by considering the output average of recent invocations. In contrast, the input size is measured programmatically by the Library Proxy component after its serialisation.

Planning and Execution

The *Domain Classifier* leverages the information provided by the Requirements Analyser to rank the best domains following a Multi-Criteria Decision Making (MCDM).

In particular, we implement the MCDM based on the SMART [76] algorithm, in which multiple competing QoS attributes are taken into account using the following equation:

$$Smart(p) = \frac{\sum_{u=1}^{U} value_u(p) * weight_u}{\sum_{u=1}^{U} weight_u} \tag{4.6}$$

Where p is a domain, the considered QoS attributes are response time and battery consumption (thus $U = 2$), and their weights are represented as explained in Section 2.5.2.

Algorithm 5 describes the main procedure used by the Domain Classifier. The algorithm first retrieves the list of available domains for that particular function (i.e., whose availability have been confirmed through A3-E Awareness). The algorithm also gets the average output size for that particular function, as well as the weight for both latency (i.e., response time) and battery (i.e., battery consumption).

Algorithm 5 A3-E Selection Algorithm

```
1: function GETDOMAINSELECTION(function, inputSize)
2:     availableDomains ← GETAVAILABLEDOMAINS(function)
3:     outputSize ← GETAVERAGEOUTPUTSIZE(function)
4:     latencyWeight ← function.getLatencyWeight()
5:     batteryWeight ← function.getBatteryWeight()
6:     maxLatency, maxConsumption, maxScore ← 0
7:     domainScores, domainUtilities ← {}
8:     for all domain ∈ availableDomains do
9:         latency ← GETRESPONSETIME(domain, function, inputSize, outputSize)
10:        if latency ≥ maxLatency then
11:            maxLatency ← latency
12:        end if
13:        consumption ←
               GETBATTERYCONSUMPTION(domain, function, inputSize, outputSize)
14:        if consumption ≥ maxConsumption then
15:            maxConsumption ← consumption
16:        end if
17:        domainQoS ← (domain, latency, consumption)
18:    end for
19:    for all (domain, latency, consumption) ∈ domainQoS do
20:        latencyScore ← latency/maxLatency
21:        consumptionScore ← consumption/maxConsumption
22:        score ←
               (latencyScore * latencyWeight) + (batteryScore * batteryWeight)
23:        if score ≥ maxScore then
24:            maxScore ← score
25:            domainScores ← (domain, score)
26:        end if
27:    end for
28:    for all (domain, score) ∈ domainScores do
29:        utility ← 0
30:        if maxScore > 0 then
31:            utility ← score/maxScore
32:            domainUtilities ← (domain, utility)
33:        end if
34:    end for
35:    domainSelection ← SORTDOMAINSBYUTILITY(domainUtilities)
36:    return domainSelection
37: end function
```

After initialising all variables, the procedure iterates over the available domains (lines $8-18$). For each domain, it obtains both the latency (line $line9$) and the battery consumption (line 13), which are compared to the current maximum of each attribute and added to the list of domain QoS (line 17).

In the next loop (lines $19-27$), the score for each attribute is computed concerning the respective maximum. The algorithm then computes the overall domain's score taking into account the respective weights for each attribute (line 22). The iteration concludes with the update of the domain score list and the max score if it applies (lines $13-15$).

The last loop (lines $28-34$) is used to normalise the domain scores using the max score value (line 31) and produce a final list of domain utilities (line 32). The latter will then be used to sort the domains and obtain the domain selection rank (line 35).

Algorithm 6 Response time computation for a given domain and function

1: **function** GETRESPONSETIME($domain, function, inputSize, outputSize$)
2: **if** $domain.type == REMOTE$ **then**
3: $throughput \leftarrow$ GETAVGLINKTHROUGHPUT($domain, function$)
4: $transferTime \leftarrow (inputSize + outputSize)/throughput$
5: $serviceTime \leftarrow$ GETAVGSERVICETIME($domain, function$)
6: $responseTime \leftarrow transferTime + serviceTime$
7: **return** $score$
8: **else**
9: $responseTime \leftarrow$ GETAVGEXECTIME($domain, function$)
10: **end if**
11: **end function**

Algorithm 7 Battery consumption for a given domain and function

1: **function** GETBATTERYCONSUMPTION($domain, function, inputSize, outputSize$)
2: **if** $domain.type == REMOTE$ **then**
3: $throughput \leftarrow$ GETAVGLINKTHROUGHPUT($domain, function$)
4: $transferTime \leftarrow (inputSize + outputSize)/throughput$
5: $netConsumption \leftarrow$ GETNETCONSUMPTION($throughput, transferTime$)
6: **else**
7: $cpuTickFreq \leftarrow$ GETCPUTICKFREQ()
8: $cpuTicks \leftarrow$ GETAVGCPUTICKS($function$)
9: $execTime \leftarrow cpuTicks/cpuTickFreq$
10: $cpuConsumption \leftarrow$ GETCPUCONSUMPTION($cpuTicks, execTime$)
11: **end if**
12: **return** $score$
13: **end function**

Algorithms 6 and 7 present the subprocedures used to respectively obtain the response time and battery consumption scores.

4.6.6 Library Proxy

To implement *Engagement*, the Library Proxy handles requests triggered by client applications by invoking the functions provided by the previously-selected domain.

The *Invocation Resolver* is the principal sub-component in the Library Proxy architecture. This component maps each domain type to a specific interface: edge and cloud domains are engaged through HTTP requests, while requests to the mobile domain are delegated to the homonym component (see Section 4.6.7).

Regardless of the domain selected, the Invocation Resolver measures for each request the following metrics: the request payload size (input size), the response payload size (output size), and the response time. Specifically, the response time is either collected from the Mobile Domain (using the method in Section 2) or from a specific field included in the response from the remote domain.

Each of the aforementioned metrics is fed into the *Performance Monitor* component. This component makes use of heuristics to decide when to report the *execution metrics* batch to the Local Feedback Manager: when a maximum number of execution metrics have been performed (currently, 25 executions); or before that upon expiration of a timer (currently, $5s$).

The Invocation Resolver harnesses the availability of multiple domains in the ranked domain selection list to provide *fault tolerance* [7] to the continuum application. More specifically, the Invocation Resolver parses the response from the currently engaged domain. If it contains an error (e.g. HTTP error from remote domains) the Invocation Resolver discards the associated execution metrics and retries the failed request with the next best domain. Domains that exhibit erratic behaviour are blacklisted for a period that grows proportionally to the number of failures. At the first successful invocation, a particular domain is removed from the blacklist.

Last but not least, the *Registration Manager* receives from continuum applications the App Config file containing all required metadata, including the fully qualified path of local functions and the repository URL of remote functions (see Listing 4.3).

The Registration Manager also keeps a callback open with the Autonomic Manager, throughout which the latter updates the Invocation Resolver with the latest *domain selection* for each of the functions required by

this continuum application.

4.6.7 Mobile Domain

Our prototype implements the mobile domain abstraction as a homonym module within the Library Proxy component.

The Mobile Domain supports two types of functions: natively supported Java functions and JavaScript functions, which require a JNI wrapper for their execution on Android devices. Note that, existing FaaS platforms support a variety of other languages. More comprehensive implementations of the Mobile Domain may leverage additional wrappers.

In contrast to cloud and edge domains, the Mobile Domain receives the fully qualified path name (e.g., a JavaScript file that implements the function), which is added to an internal function registry upon application registration. During Engagement and upon the selection of this domain, the Mobile Domain handles *function request* events are by looking up for the corresponding functions. Once found, the function is called with the parameters in the original C-request, and produces a *function response* with the execution output.

CHAPTER 5

The PAPS Framework

5.1 Overview

Edge computing promises many benefits. Nonetheless, the management of geo-distributed infrastructures poses significant challenges: not only one must provision and allocate computational resources to components, but these components must be placed onto the different nodes by taking into account propagation latency along with the availability of resources.

The monitoring and analysis of the current workload, availability of resources, and performance of components is key for the efficient the allocation of resources and placement of components, but it must be carried out in a timely manner for the entire system. Network latency and time-consuming decisions may jeopardise the effectiveness of the decision process, especially with highly volatile workloads —a likely-to-happen scenario with densely distributed edge nodes that serve the needs of mobile/IoT devices.

The placement and migration of services in MEC and similar edge-centric architectures received considerable attention from researchers. Optimal solutions —e.g. that minimise network latency or maximise resource utilisation— are typically NP-Hard [124]. Many works tackled variations

of the allocation and placement problems with heuristics [71, 114, 132] and approximations [122, 124, 125]. Although some report the complexity of their solution, the majority resort to experiments. In many cases, evaluation is carried with a reduced number of nodes and components (or services). Moreover, few works take into account the challenges related to the monitoring, analysis, and actuation of decisions taken by a centralised, omniscient orchestrator. Hence, a more holistic framework targeting the management of geo-distributed infrastructures and services is still missing.

In this chapter, we propose *PAPS* — ***P****artitioning*, ***A****llocation*, ***P****lacement*, and ***P****rovisioning* — a framework for tackling the effective and efficient management of geo-distributed infrastructures and services through decentralized self-adaptation and serverless computing.

In contrast with Chapter 4, in which we focused on the self-management of the life-cycle of functions opportunistically deployed to mobile devices and disjoint edge and cloud infrastructures, herein we move our attention to the self-management of geo-distributed infrastructures and services from a single provider, namely a MEC operator. While in Chapter 4 we assumed that disparate *edge domains* to operate independently, the PAPS framework is based on the assumption that lateral collaboration [77] among edge nodes is critical for the system robustness, performance, and scalability.

The remainder of this chapter is organised as follows. Section 5.2 discusses the challenges of managing decentralised infrastructures and services. Section 5.3 presents the system model and presents a formulation of the self-management problem tackled by our framework. Next, Section 5.4 presents the PAPS framework, whereas Section 5.5 describes the implementation of PAPS as a simulation platform.

5.2 Management Challenges

The challenges of managing geo-distributed infrastructures and services may be approached from multiple angles.

From the bottommost level, cloudlets [97] and surrogate MEC nodes [105] may employ self-managing techniques [47, 81] to augment the capabilities of transient mobile and IoT devices. This approach matches the dense distribution of infrastructure with full decentralisation of control over the allocation of resources and the provisioning of various services.

The A3-E framework presented in Chapter 4 instantiates this approach. Moreover, A3-E leverages the serverless architecture introduced in Chapter 3 to promote the accessibility and efficiency in which Self-Managed Computing Services are provided to latency-intensive and data-intensive

applications hosted by IoT and mobile devices.

Moving up in the decentralised infrastructure hierarchy —to the fog layer [77]— one can imagine several ways to realise management. For instance, through dedicated control units deployed within or in proximity to the radio access network [59, 123] orchestrating several MEC nodes; or even through supernodes (or leaders) responsible for the surrounding and underlying sections of the topology. At last, one can also think of powerful and omniscient —yet physically remote— centralised orchestrators [56] hosted by cloud data centres.

The question that arises is: *what are the benefits and limitations of tackling the management of geo-distributed infrastructures and services at each of these three levels?*

The answer to the question above requires the specification of what kind of management decisions needs to be tackled.

If we consider the provisioning of computing services needed for the execution of serverless functions, the management decision is twofold. First, it concerns the number of containers that need to be allocated for each function given the workload and SLA. Further, it concerns where in the topology the containers must be placed to satisfy the demand coming from various access points in the topology while satisfying the maximum acceptable delay. In light of this formulation, the primary benefits and limitations of tackling the management challenges at each level are as follows.

At the node level, data locality yields maximum responsiveness at the cost of limited processing power and reduced visibility over the topology.

At the regional level, micro data centres [56] and Virtualised Network Infrastructure [102] offer the computational power needed for more complex decisions and have a regional view of the system, but might not be able to monitor and decide so quickly as localised, self-managing nodes.

Finally, a centralised orchestrator may harness virtually infinite resources from cloud data centres but are too far away to perform fine-grained management decisions for the whole topology in a timely and effective manner.

We took each of the previous aspects into consideration in the design of PAPS, a holistic framework targeting the management of densely distributed edge infrastructure hosting the execution of serverless functions.

5.3 System Model

5.3.1 Infrastructure Model

While many edge-centric architectures have been proposed, in this chapter we shall focus on a MEC topology composed of a finite set of nodes

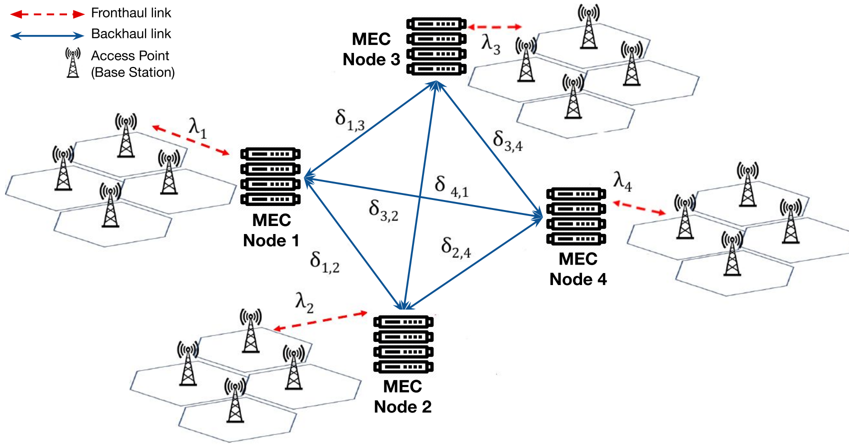


Figure 5.1: *A topology formed by geo-distributed MEC nodes; each node i is reachable by edge devices connected to a base station in its area through the fronthaul network (delay λ_i), or through the backhaul network (delay $\delta_{i,j}$)*

$\mathcal{N}$. Figure 5.1 presents such a topology. Mobile and IoT devices access the system through cellular base stations. Each device is connected to a MEC node $i \in \mathcal{N}$ through the *fronthaul network*. MEC nodes in $\mathcal{N}$ are interconnected through the *backhaul network*. The total propagation delay $D_{i,j}$ between a client device accessing the system through the base station co-located with the MEC node $i \in \mathcal{N}$ and served by the MEC node $j \in \mathcal{N}$ is defined as:

$$D_{i,j} = \begin{cases} \lambda_i + \delta_{i,j}, & \text{if } i \text{ != } j \\ \lambda_i, & \text{if } i = j \end{cases} \tag{5.1}$$

where λ_i and $\delta_{i,j}$ are respectively the propagation delay of the fronthaul and backhaul networks.

5.3.2 Function-as-a-Service

Our framework targets the dynamic allocation and placement of containers required for the execution of serverless functions [9]. Even if the allocation model adopted by serverless vendors can vary [58], typically functions are given access to a fixed *CPU share* proportional to their memory capacity, similarly to the leading Function-as-a-Service (FaaS) offerings.

A limit (e.g. $250MB$) is imposed on the function package size. The latter restricts time needed to load the function and dependencies into a fresh (cold) container. It is also paramount to cope with the storage limitations

of densely distributed edge nodes.

Additionally to the *memory capacity* —which specifies the maximum memory available to the container hosting the function execution— we assume a function deployment descriptor to be specified with a *Service-Level Agreement* (SLA) between the MEC operator and the customer deploying the function (e.g. an application provider).

To the best of our knowledge, the SLA specification is not supported by existing FaaS vendors [4, 34, 43, 67]. To better support applications with strict requirements for latency, the PAPS framework takes into account *response time* as an SLA attribute. More precisely, the SLA is specified with two attributes. The *Response Time* (RT_{SLA}) defines the upper limit for the round-trip time between the arrival of a request in the system (through one of the access points across the topology), its execution, and its exit through the same access point (synchronous invocation); or from the arrival of request until the end of function execution (asynchronous invocation). Table 5.1 illustrates two possible SLAs.

Latency-sensitive	**Data-intensive**
Response Time: $\leq 120ms$	Response Time: $\leq 500ms$
Maximum Execution Time: $90ms$	Maximum Execution Time: $380ms$
Invocation Mode: $synchronous$	Invocation Mode: $asynchronous$

Table 5.1: *Examples of Response Time SLA*

Each SLA is specified with two attributes.

The *Response Time* (RT_{SLA}) specifies the upper limit for the round-trip time between the arrival of a request in the system (through one of the access points across the topology), its service, and the response exit through the same access point (synchronous invocation); or from the arrival of request until the end of function execution (asynchronous invocation).

The *Maximum Execution Time* (E_{MAX}) limits the function execution time. The latter is a common attribute in cloud-based FaaS platforms [4, 104]. In the context of our framework, it is key to guide the allocation and placement decisions, as formulated in the following section.

5.3.3 Management Problem Formulation

For a given topology $\mathcal{N}$ and a set of admitted functions $\mathcal{F}$, the management problem is twofold: (i) to decide *how many* containers are needed for each function; and (ii) to decide *where* (onto which nodes) should each container

be placed.

Each allocated container works as a server for a specific function $f \in \mathcal{F}$, whose memory capacity is defined by m_f, whereas M_j defines the overall memory capacity of node $i \in \mathcal{N}$.

While most FaaS vendors claim to allocate resources on a request basis, queuing requests for a short period may significantly reduce the use of resources. Thus, the perceived response time is given by the following general equation:

$$RT = D + Q + E \tag{5.2}$$

where D represents the total *propagation delay*, Q the *queuing time*, and E the *execution time*.

The propagation delay is affected by the container placement decision, accordingly to Eq. 5.1. The *sojourn time* (i.e., Q + E) depends on the number of containers serving each function. Intuitively, adding containers decreases queue time to the point that all incoming requests find a warm container, i.e., to the point that $Q = 0$ and $RT = D + E$.

To minimise the use of resources, the MEC operator should control D (through placement) and Q (through scaling) so that:

$$D + Q + E \leq RT_{SLA} \tag{5.3}$$

The higher the propagation delay component, the less margin is left for queuing requests. At one extreme level, $D = RT_{SLA} - E$ and the request must be served immediately, i.e., Q must be nullified.

Conversely, serving requests at the nearest node minimises component D, but might result in poor workload distribution: some MEC nodes may become overloaded while others remain idle.

The propagation delay of D can easily become part of an optimal container placement problem formulation. On the other hand, it is less trivial to incorporate Q and E as part of a joint allocation and placement formulation.

One possible approach is to use a function to map the workload to the corresponding allocation (i.e., number of containers) needed to keep the sojourn time below a threshold. Indeed, some authors (e.g. [138]) assume the workload to follow a known probabilistic distribution; queue theory is then used to calculate the expected sojourn time.

As it will be discussed in Section 5, we refrain from assuming a known probabilistic distribution for the workload. Instead, the current sojourn time

is measured at each node for the functions it is currently hosting. This information is then used to calculate the number of containers to satisfy the workload from each source across the MEC topology and find the optimal placement solution. We, therefore, postpone the formalization of the placement and allocation problem tackled by our framework to the next section.

5.4 The PAPS Framework

In this section we present *PAPS* — ***P**artitioning, **A**llocation, **P**lacement and **S**caling* — a framework for tackling the decentralised self-management of large scale edge systems. It is comprised of three levels: the system-level, the community-level, and the node-level self-management.

5.4.1 System-Level Self-Management

Objective: The system-level self-management aims to tackle the complexity of managing the larger scale geo-distributed infrastructure by partitioning it into *delay-aware network communities*.

Community-based Partitioning

In complex networks, a network is said to have *community structure* if its node can be easily grouped into (potentially overlapping) sets of nodes such that each set is densely connected internally [129].

In our framework, we extend the previous definition and refer to a *delay-aware network community* as a set of logically interconnected MEC nodes whose network latency from one another is below a threshold.

A delay-aware network community provides a reduced space in which the solution for the placement and allocation satisfying Equation 5.3 can be computed (effective analysis). Furthermore, it allows the parallelisation of the self-adaptation process and its localisation within a geographic area (effective monitoring, analysis and actuation).

The community identification may follow different approaches. In our framework, a centralised coordinator (hereafter referred to as the *supervisor*) has a global view of the system and makes use of a *community search algorithm*. The algorithm takes two parameters: the *maximum inter-node delay* (MID) and the *maximum community size* (D_{MAX}).

The MID parameter is used to produce a sub-graph. Each of its vertices maps to a node in the MEC topology; an edge exists between two vertices if the network delay between their respective nodes is below the maximum inter-node delay.

The D_{MAX} parameter limits how many MEC nodes can belong to a community. Therefore, it is useful for limiting the complexity of the community-level self-management.

The produced sub-graph will then serve as the input to a community detection algorithm. In particular, we adopt the *SLPA* method [128], whose complexity is $O(t * n)$, where t is a predefined maximum number of iterations (e.g. $t \leq 20$) and n is the number of nodes. The SLPA method ranked first among the community detection algorithms in the literature [129].

Community Structure Adaptation

MEC nodes are co-located with fixed infrastructures. Thus, we assume that nodes and inter-node delay are expected to remain stable. Nonetheless, topological changes caused by catastrophic failures, system upgrades, and other eventualities may require the adaptation of the community structure. The primary goal of the supervisor is, therefore, to assure that communities remain consistent in their size, membership and inter-node delay.

While defining the best approach to tackle the community structure adaptation, we took into account the amount of information that needs to be monitored, as well as the complexity of the community search procedure.

The presence and health of the MEC nodes across the topology can be obtained through light-weight *heartbeat* messages sent by each node to the supervisor. This approach is commonly adopted in distributed systems of different scales and does not treat the scalability of the proposed solution.

As previously mentioned, the time complexity of the adopted community search algorithm is linear ($O(t*n)$), hence tractable even for very large topologies. Authors also suggest a modest value ($t = 20$) for the maximum number of iterations needed to find edible quality communities [128].

All these aspects favoured the choice of the supervisor as the managing entity responsible for the community structure adaptation.

The supervisor harnesses its global system view to tackle the adaptation of the community structure. More specifically, the system-level adaptation is modelled as a master-slave MAPE loop [127] depicted in Figure 5.2. While *Monitoring* and *Execution* are performed by each and every MEC node in the topology, the supervisor is in charge of the *Analysis* and *Planning* activities.

5.4.2 Community-Level Self-Management

Objective: the community-level self-management aims to assure that the MEC nodes in that community are operating under feasible conditions, i.e.,

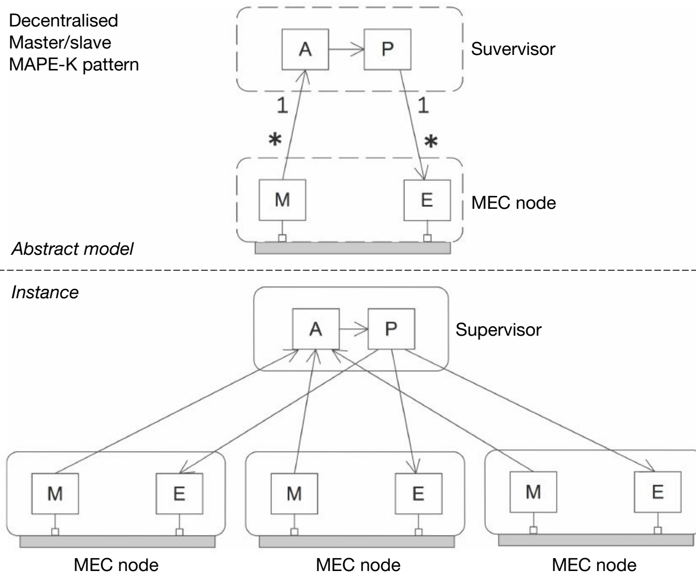


Figure 5.2: *Community structure adaptation modelled as a decentralised master/slave MAPE loop (adapted from [127])*

it aims to minimise the likelihood of SLA violations to occur and, if they occur, react in order to bring the MEC community back to its equilibrium.

Inter-Community Allocation

A first challenge that emerges when the MEC system is partitioned into communities regards the allocation of shared member resources. In other words, we must decide the share that each overlapping community will get from the shared member resources.

A trivial but less efficient approach would be to privilege one community with all resources. However, the disadvantaged communities might need more resources to cope with the additional workload while the common member is underutilised. Changes to the workload are expected to happen frequently and without warning. In order to prevent SLA violations, resources from common nodes need to flow from one overlapping community to the other. Our framework tackles this goal by weighting the aggregate demand and capacity in each overlapping community.

The aggregate demand refers to the number of containers needed to cope with the aggregate workload. The latter refers to the rate of requests incoming from base-stations co-located with MEC nodes whose network latency to the common node is below the inter-node delay threshold D_{MAX}, plus

a proportional demand share from the base station(s) co-located with the common node itself. The aggregate capacity refers to the sum of resources from the previous nodes, excluding the common node. A share of capacity is then allocated by the common node to each overlapping community proportionally to their aggregate *demand-capacity ratio*.

Algorithm 8 depicts our inter-community allocation approach. This procedure is greedily performed for all MEC nodes in the topology that belongs to two or more communities.

Algorithm 8 CapacityDemandRatio(C, s, D_{MAX})

1: $neighborsInRange \leftarrow$ GETNEIGHBORS(C, s, D_{MAX})
2: $aggDemand \leftarrow 0,\ aggCapacity \leftarrow 0$
3: **for all** $n \in neighborsInRange$ **do**
4: $\quad aggDemand \leftarrow$ GETDEMAND(n)
5: $\quad aggregateCapacity \leftarrow$ GETCAPACITY(n)
6: **end for**
7: $ovCount \leftarrow$ GETOVCOUNT(s)
8: $demandShare \leftarrow$ GETDEMAND(s) / $ovCount$
9: $aggDemand \leftarrow aggDemand + demandShare$
10: **return** $aggDemand$ / $aggCapacity$

Intra-Community Allocation and Placement

Analogously to the inter-community allocation, the intra-community allocation aims to distribute resources among nodes within a community given the aggregate demand and capacity in that community.

Within each community, a leader is responsible for solving the allocation and placement problem introduced in Section 5.3.3. Such a *centralization within decentralization* has the following advantages: (i) it allows the optimal placement problem to be solved in a single step for the whole community; (ii) it eliminates the need for a more complex coordination protocol. More importantly, the leader-based approach allows the placement problem to be solved by well-known centralised optimisation techniques.

A problem that arises from the complexity of the container placement problem is high-resolution time, which prevents communities to adapt to workload fluctuations on time. Until a placement solution is computed, the workload may have changed, requiring a new solution.

Pro-active adaptation [70, 89] could be used to mitigate the aforementioned problem. For instance, if the arrival of requests is characterised by a well-known probabilistic distribution (e.g. Poisson), the container allocation procedure might then benefit from queue theory to predict the number

of containers that are needed to keep the sojourn time in Eq. 5.2 at a specific value and thereby jointly solve the allocation and placement problem.

To our disadvantage, the decentralised infrastructure model makes the previous assumption less realistic. Not only users will freely enter and exit different areas, but the aggregate workload to be served by each node is limited compared to typical cloud services. Even a small group of users that enters or exits a MEC node area may cause significant workload variation.

In light of this, our framework favours reactive adaptation for solving the allocation and placement problem. Our solution draws inspiration from the Ultra-Stable system architecture [81].

The community-level self-management acts as the secondary feedback loop in the Ultra-Stable system. When the workload fluctuations are significant enough to threat or to throw the node-level self-management out of its limits, the community-level self-management provides the community with a new allocation and placement solution.

In turn, the node-level self-management works as the primary feedback loop in the Ultra-Stable system architecture. Through its sensors, the MEC node monitors subtle changes in the environment (i.e., in the actual workload for each function). It accordingly responds, through its effectors, by changing the node-level allocation (i.e., the number of containers per function).

Hence, the community-level placement targets not a single solution, but a solution space in which the node-level container scaling ultimately takes place. In more details:

- The optimal placement solution determines the *target* amount of containers to be hosted by each community member. It considers the *aggregate workload* incoming from all access points in the community.

- Load balancers in the community infrastructure use the placement solution to route the workload incoming from different sources (i.e., base stations) to their respective destinations (i.e., MEC nodes).

- Each member (MEC node) in the community has the autonomy to decide the actual number of containers it hosts per placed function based on the *actual workload* and the resources available.

- As the workload fluctuates, response time deviates from its target value; the node-level self-management takes care of the timely scaling of containers to optimise resource usage while preventing SLA violations.

- The optimal solution is enforced by each community member in case of local resource contention.

Using a more recent reference model [47], the community-level self-management consists of an instance of the *regional planner* MAPE loop [127] depicted in Figure 5.3.

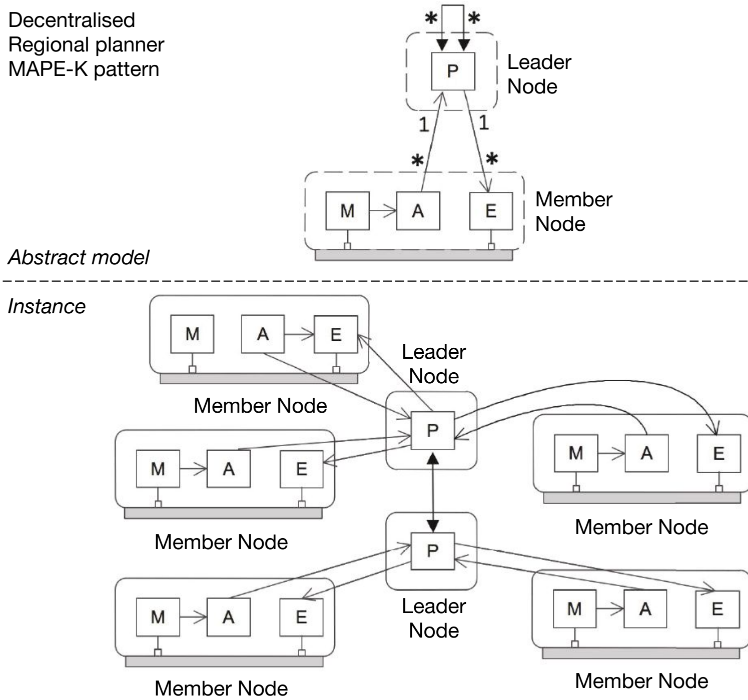


Figure 5.3: *Community-level self-management modelled as a regional planner MAPE loop (adapted from [127])*

Each community member takes advantage of its privileged position within the MEC topology to *monitor* and *analyse* the workload coming from adjacent base stations (see Fig. 5.1).

The number of containers needed to cope with a given workload while satisfying the function SLA is determined at the node level using a control loop with a short period —compatible with the container start-up time (i.e., up to a few seconds).

In turn, the community leader utilises information to *plan*, through the transfer function in Section 5.4.3, the number of containers needed to satisfy the response time SLA. The transfer function takes as input the aggregate workload over a larger control period —compatible with the time

needed to compute the optimal placement solution (i.e., up to a few minutes, depending on the formulation, the number of nodes in the community, and the number of admitted functions).

At last, the community-level adaptation is *executed* by each affected node in the community with the update of its targeted placement and allocation for each admitted function. Depending on how the new placement solution diverges, the MEC node may have to add or remove function(s) from/to its catalogue.

Optimal Container Placement. The PAPS frameworks is agnostic with respect to the placement optimization formulation. Herein, we formulate it as a Mixed Integer Programming (MIP) problem as follows:

$$\min_{x} \quad \sum_{i\in\mathcal{N}}\sum_{j\in\mathcal{N}}\sum_{f\in\mathcal{F}} d_{i,j} * x_{f,i,j} \tag{5.4a}$$

$$\text{subject to} \quad x_{f,i,j} * d_{i,j} \leq x_{f,i,j} * D_k \qquad \forall i \in \mathcal{N}, \forall j \in \mathcal{N}, \forall f \in \mathcal{F} \tag{5.4b}$$

$$\sum_{i\in\mathcal{N}}\sum_{f\in\mathcal{F}} c_{f,i} * m_j * x_{f,i,j} \leq M_j \qquad \forall j \in \mathcal{N} \tag{5.4c}$$

$$\sum_{j\in\mathcal{N}} c_{f,i} * x_{f,i,j} = c_{f,i} \qquad \forall i \in \mathcal{N}, \forall f \in \mathcal{F} \tag{5.4d}$$

where $x_{f,i,j}$ is a continuous decision variable that denotes the fraction of the container demand $c_{f,i} \in \mathbb{N}_{>0}$ for function $f \in F$ from source node $i \in \mathcal{N}$ that should be hosted by node $j \in \mathcal{N}$. The rationale behind the use of a continuous variable is as follows.

The control-theoretic container scaling in Section 5.4.3 calculates the ideal allocation per function per workload source ($c_{f,i}$). Although one can not scale a non-integral number of containers (e.g. scale-up half container), the ideal allocation from various sources may be combined. Indeed, server consolidation is commonly used to boost resource usage efficiency. It is also the default behaviour of the MIP solver.

Conversely, the ideal allocation from a single source (e.g. 4 containers) may be distributed among two or more nodes. This is especially so if node capacity is insufficient. The continuous variable $x_{f,i,j}$ enables the MIP solver to provide such type of solution.

The objective function in Eq. 5.4a minimizes the overall network delay resulting from the container placement. The first constraint (Eq. 5.4b)

limits the network propagation delay. Since $x_{f,i,j}$ is a continuous variable, it appears at both sides so that a fractional value is counterbalanced at the threshold side. Specifically, D_k is calculated using the following equation:

$$D_f = \beta * (RT_{SLA,f} - E_{MAX,f}) \tag{5.5}$$

where $0 < \beta \leq 1$ defines the fraction of the marginal response time $RT_{SLA,f} - E_{MAX,f}$ for function $f \in F$ that can be used for networking. Conversely, the complement $(1 - \beta)$ defines the fraction of the marginal response time used for queuing function f while hosted by node j:

$$Q_{f,j} = (1 - \beta) * (RT_{SLA,f} - E_{MAX,f}) + (E_{MAX,f} - E_{f,j}) \tag{5.6}$$

where $E_{f,j}$ is the monitored execution time for function f hosted by node j. The queue component $Q_{f,j}$ is particularly important to the control-theoretic container scaling (see Section 5.4.3).

The second constraint (Eq. 5.4c) assures the number of containers placed at a node $j \in \mathcal{N}$ does not violate its memory capacity M_j.

Finally, the last constraint (Eq. 5.4d) assures that all required containers are placed.

5.4.3 Node-Level Self-Management

Objective: The node-level self-management aims to efficiently and effectively orchestrate the scaling of containers needed to satisfy the response time SLA of each admitted function given the fluctuations in the workload and the target placement and allocation defined by the community leader.

Given a fixed allocation of resources, the response time of a function can change due to various reasons, including:

- Variations in the request arrival rate (i.e., due to mobility and churn).
- Variations in the execution time (e.g. due to input variations).
- Disturbances in the execution environment (e.g. at the operating system or hardware level).

While some factors are harder to quantify and account for, others are possible to monitor and take into account in determining the appropriate allocation needed to prevent violations of the response time SLA. Our framework leverage a control-theoretic approach [10] proposed in Chapter 4 to

solve the node-level container scaling problem. We recall the more essential aspects of its operation regarding the PAPS framework; the reader can find more details in Section 4.5.4 where it is introduced.

Contro-theoretic Container Scaling

The control system is responsible for the deployment of containers to the cluster of (virtual) machines composing the MEC node pool of resources, from now on referred to as the *control plant*.

The control plant is subject to different signals that could be measurable (input variables) or unknown (disturbances). Considering a discrete time, for each admitted serverless function we define $\lambda(k)$ as the function of the measured arrival rate of requests at each control time k; while $\bar{\lambda}(k)$ is the corresponding vector for all admitted functions.

At time k, the function is executed in a $c_j(k)$ number of allocated containers. The disturbances are defined as $\bar{d}$ and cannot be directly controlled and measured. Finally, $\bar{\tau}$ is the system output and corresponds to the response time vector comprising all functions, whereas $\bar{\tau}^{\circ}$ corresponds to the vector of desired response time per function (or control *set-point*), according to the SLA of each function.

In our current set-up, the function $\bar{\tau}^{\circ}(k)$ does not vary over time, meaning a constant targeted response time for each function. Of course, these values should be set below the SLA threshold. Moreover, since response time cannot be measured instantaneously but by aggregating the execution time of different requests over a predefined time window, many aggregation techniques could be used without any change to the system model and controller. In our framework, we compute the average of the response time values in $\bar{\tau}$ within each control period, but stricter aggregation functions, such as $99th$ percentile, could be used depending on the needs of the MEC operator.

Our framework dedicates one controller per function. A characteristic function is used to model the system with enough details to govern the dynamics of the plant. We assume that this function does not need to be linear but regular enough to be linearisable in the domain space of interest. Moreover, we consider this function to be dependent on the ratio of the number of allocated containers c and the request rate λ.

The characteristic function monotonically decreases towards a possible lower horizontal asymptote, as it can be assumed that once the parallelism degree of a function is fulfilled by the available containers, adding new ones entails no further benefits in terms of response time. We found a practically acceptable function to be:

$$f\left(\frac{c(k)}{\lambda(k)}\right) = \widetilde{u}(k) = c_1 + \frac{c_2}{1 + c_3\frac{c(k)}{\lambda(k)}} \tag{5.7}$$

where parameters c_1, c_2, and c_3 were obtained through profiling of each function.

As control technique, we rely on PI controllers because they are able to effectively control systems dominated by a first order dynamic [6] (i.e., representable with first order differential equations) such as the studied ones.

At each control step, function controllers run independently (i.e. without synchronization) to compute the next container scale for the corresponding function, which is added to the vector $\hat{c}$. The number of containers in $\hat{c}$ is not immediately actuated since the sum of required containers could be higher than the entire capacity of the MEC node. Instead, $\hat{c}$ is passed to a contention manager. This component outputs a vector $\bar{c}$, which is the actual number of containers per function, defined as:

$$\bar{c}(k) = \begin{cases} \hat{c}(k), & \text{if } \textit{no resource contention} \\ scaleDown(\hat{c}(k)), & \text{otherwise} \end{cases} \tag{5.8}$$

where function $scaleDown$ scales down the values in $\hat{c}$ according to the thresholds defined by the placement solution by the community leader (see Section 5.4.2). The contention manager then updates the state of each controller to be consistent with the actual allocation.

5.5 PAPS Simulator

In this section, we briefly describe the implementation of the PAPS Simulator [1]. The purpose of the simulator is twofold. First, it was used to assess the feasibility of our framework. Second, we aim to provide a flexible and accessible simulation environment to be used with various topologies, workload, and application scenarios. The simulator also allows experiments to be performed with variations of the optimal placement and allocation formulation. Also importantly, a rich set of parameters allows experiments to be performed with a broad range of scenarios, which include turning on and off the control-theoretic container scaling at the node level.

[1] Source code accessible from `https://github.com/deib-polimi/PAPS`

5.5.1 PeerSim

We have implemented our framework using PeerSim [69], a peer-to-peer (P2P) simulator framework.

PeerSim is the result of an academic effort to produce a scalable peer-to-peer simulation framework. Several factors motivated our choice of PeerSim as the baseline simulation framework for implementing PAPS.

First and foremost, we took into account the distributed nature of PAPS. Differently from other works that abstract away the implementation of a proposed solution for the management of complex edge systems, PAPS focuses on the inherent challenges of monitoring, analysing, planning, and executing adaptation decisions for the entire MEC topology.

In the literature, one can find simulation platforms for edge computing. EdgeCloudSim [111] and iFogSim [36] are two examples of simulation platforms targeting the orchestration of resources from an edge (or fog) system. These platforms extend the popular cloud simulator framework CloudSim [19]. CloudSim features a centralised architecture. It is particularly suited for the evaluation of load and data distribution but does not model with details important aspects of communication.

In contrast, PeerSim provides a rich set of features for the implementation of distributed protocols, as well as utilities for the generation of dynamic workload and the exchange of messages using reliable or unreliable channels, with or without delay. It also provides out-of-the-box support for the creation of various sort of flat or hierarchical topologies, which can also vary over time. Also importantly, PeerSim is designed with scalability as a first-class requirement.

We harness the features offered by PeerSim to implement the separate self-management layers of our framework as P2P protocols, which then can be evaluated with a more realistic simulation of a large scale MEC topology. Additional comparison with other simulation platforms in the literature can be found in Section 7.3.

5.5.2 Implementation Overview

The PAPS Simulator architecture reflects the underlying model used by PeerSim. Thus, before describing each main component in our solution, we briefly introduce the corresponding PeerSim class or functionality.

The *Protocol* class is the main component in PeerSim. As the name suggests, this class is used to implement the behaviour to be performed by the nodes that implement a given protocol. Instances of the Protocol class are cloned into every node in the topology. Each node then executes

the protocol once at every cycle (cycle-based simulation model) or every scheduled event (event-based simulation model).

In its current version, the PAPS Simulator implements two protocols: a *Community Protocol* and the *Node Protocol*. Each protocol respectively implements the behaviour of MEC nodes as community members (i.e., as a leader or a common member) and individual nodes. The Community Protocol implements the adaptation activities from the regional planner MAPE loop described in Section 5.4.3 is implemented. In turn, the Node Protocol implements a part of the node-level adaptation and delegates to a library the implementation of the control-theoretical container scaling.

Another important PeerSim class is *Control*. In contrast with the Protocol class, the Control class is instantiated once per simulation and is not associated with a particular node. The purpose of this class is to enable the implementation of custom operations performed over the nodes or any other entity that compose the implemented simulation, be it once (e.g. at initialisation time) or throughout the simulation (i.e., at every cycle or event).

The PAPS Simulator extends the Control class to initialise the state of the community protocol (*CommunityStateInitializer*) and node protocol instances (*NodeStateInitializer*). It is also used to initialise the inter-node delay and the memory capacity of each node, as well as the catalogue of functions admitted into the system.

Moreover, our implementation leverages the PeerSim's Control to initialise the workload coming from different sources across the topology and vary the workload throughout the simulation (*WorkloadFluctuation*). Finally, the Control class is also extended to synchronise the logical time of the simulation with the real function execution time (see the Control-Theoretic Container Scaling below).

Our simulation platform leverage the utility classes provided by the simulation framework to build the MEC topology. Later on, the nodes in the topology can retrieve the list of links and thereby the nodes connected to them. PeerSim also helps with the implementation of the transport layer. A simulation may be built with reliable or unreliable transport. In the latter case, a parameter defines the message drop probability. This feature is important for the design of simulation scenarios in which unreliable channels are used by leaders to collect metrics from community members and to inform them about adaptation decisions.

At last, we extended PeerSim's *Node* class to characterise nodes in the topology regarding their essential attributes and state (*FogNode*). Differently from *node-centric* P2P simulators, PeerSim enforces the decoupling of the P2P state and behaviour (protocol class) from the intrinsic attributes

and behaviour that characterise each node.

Our implementation extends the Node class to define the attributes that characterise a MEC Node, namely its computational capacity. This class may be further extended to include additional MEC node features such as its hierarchical position in the topology.

Optimal Container Allocation and Placement

We have modelled the optimal placement of containers onto MEC nodes within each community using IBM CPLEX[2], a state-of-the-art solver for linear programming, mixed-integer programming and quadratic programming. Our implementation interfaces with CPLEX through its command-line interface (*OPLEX*).

At each iteration of the community-level feedback loop, the community leader performs an asynchronous call to the solver CLI, which must be installed in the system that hosts the simulation execution.

The call to the solver CLI includes the path to two files: the *model* file, in which the container placement is formulated (in our solution, as a mixed integer programming problem) and an additional *data* file containing the initialisation of the (non-decision) variables in the formulation (e.g. MEC node capacity, inter-node delay).

Our solution adopts a template-based approach in which both files are built at each community-level iteration based on a given skeleton of the model and data files. The classes in the *solver* package encapsulate the interfacing with the solver of choice. Future implementations may either extend or include new classes for variations of the optimisation problem formulation and corresponding variables.

Once the optimisation is solved, results are collected via standard output and parsed to obtain the corresponding allocation and placement solution, i.e., the targeted number of containers per admitted function to be hosted by each MEC node in the community.

Control-Theoretic Container Scaling

Our solution employs a modular architecture to separate the main simulator component from the library[3] that implements the control-theoretic method for container provisioning at the node level.

The node library was implemented as a dynamic thread pool, where one container is a thread that executes the incoming requests. The maxi-

[2] https://www.ibm.com/analytics/cplex-optimizer

[3] Source code accessible from `https://github.com/deib-polimi/ppap-node`

mum number of containers that can be allocated onto a node depends on its memory capacity and the memory requirements of the functions to be deployed (respectively, 96GB and 128MB in our experiments).

Nodes in the topology instantiate the library above, which in turn exposes the node its single interface (*NodeFacade*). The interface includes methods for adding/removing functions and for updating the targeted allocation, as decided by the community-level container placement.

The control library also includes functionality for mimicking the real execution of functions using Java threads, based on which the control system measures the aggregate response time of various functions. The library interface also exposes a method for triggering the execution of a function. The latter is used to transfer the responsibility for the workload generation from the node library component to the main simulator thread.

Following the technique in Section 5.4.3, the *Controller* component decides the number of containers needed to cope with the function SLA given the monitored response time and workload. In the case of resource contention, the number of containers is scaled down according to the target value decided by the community-level container placement.

Last but not least, the NodeFacade includes a static method for retrieving the number of containers for a given workload based on the characteristic function used by the control system (see Section 5.4.3). This interface method is consumed by the Community Protocol to retrieve the *container demand* given an *aggregate workload* as part of community-level feedback loop described in Section 5.4.2.

CHAPTER 6

Evaluation

6.1 Serverless MEC Architecture

6.1.1 Overview

In this section, we report on the evaluation of the Serverless MEC Architecture. First, the architecture is instantiated with the MAR application described in Section 3.2.4. The primary goal is to demonstrate the use of the platform services with a real application scenario and to assess the platform from the development perspective. Secondly, we performed a series of experiments to demonstrate key attributes of the proposed architecture, such as the satisfaction of targeted use case requirements and scalability.

6.1.2 Application Scenario: Mobile Augmented Reality

The MAR application appears in many technical [41,103] and research [16, 126] publications as one of the use cases that would significantly benefit from a MEC architecture. MAR distinguishes from typical Augmented and Virtual Reality applications as it leverages the mobility offered by smartphones, smart glasses, and other modern pervasive devices. Hence, MEC services are paramount for the seamless operation of these applications as

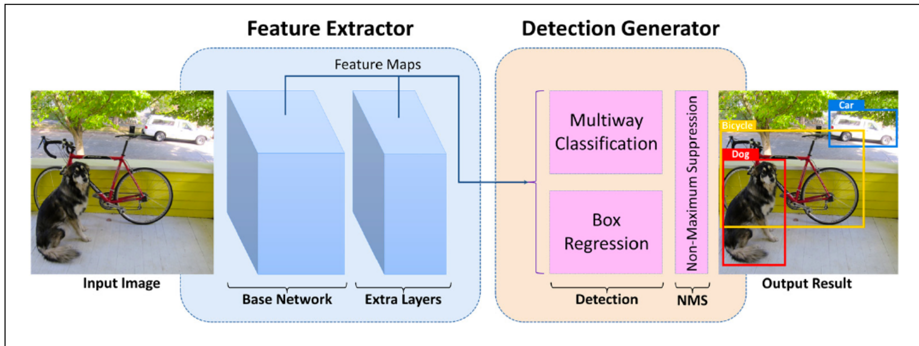


Figure 6.1: *An overview of the SSD object detection method. Image adapted from [84].*

users enter and exit different city areas.

Object Detection Method

At the heart of the MAR application is the method for identifying objects composing the captured scenes. In the literature, one can find several methods for object detection [84]. In the particular case of outdoors augmented reality like our MAR application, a combination of multiple sensors is required for robust detection, such as a GPS receiver for locations and distances and inertial and magnetic sensors for orientations. Additionally, it is a common practice to employ computer-vision techniques to accomplish more robust object detection and tracking [84]. Among these, natural feature detection methods were successfully ported into mobile devices [42, 118]. In some approaches, a client-server architecture is employed to offload heavyweight computation, which poses the challenge of limiting the impact of network latency on Quality-of-Experience.

More recently, the application of machine-learning techniques to computer-vision and object detection attracted considerable attention from both researchers and practitioners. Instead of relying on manual feature engineering, deep learning methods automatically discover from raw data the representations needed for classification or detection, thereby taking advantage of the increasing availability of computational resources and data [84]. Among these methods, some are capable of achieving real-time performance with significant detection accuracy [57,85]. Accordingly, we demonstrate our Serverless MEC Architecture with the MAR application using the *Single Shot Detector* (SSD) [57], a state-of-the-art object detection method based on deep-learning.

Figure 6.1 provides an overview of the SSD method. The first step

Listing 6.1: *Object detection function.*

```
1  import numpy as np
2  import cv2
3  import base64
4  import time
5
6  def main(args):
7      global cache
8      if 'cache' not in globals():
9          neural_network = cv2.dnn.readNetFromCaffe("./imageRecognition/
               MobileNetSSD_deploy.prototxt.txt","./imageRecognition/
               MobileNetSSD_deploy.caffemodel")
10         cache = neural_network
11     else:
12         neural_network = cache
13
14     classNames = ["background", "aeroplane", "bicycle", "bird", "boat", "
           bottle", "bus", "car", "cat", "chair", "cow", "diningtable", "dog"
           , "horse", "motorbike", "person", "pottedplant", "sheep", "sofa",
           "train", "tvmonitor"]
15     image = cv2.imdecode(np.fromstring(base64.b64decode(args["image"]),
           dtype=np.uint8),1)
16     input_blob = cv2.dnn.blobFromImage(image, 0.007843,(300, 300), (127.5,
            127.5, 127.5), False, False)
17     neural_network.setInput(input_blob)
18     detections = neural_network.forward()[0][0]
19     contents = []
20     for elem in detections:
21         name = classNames[int(elem[1])]
22         confidence = 100 * sorted(elem[2:], reverse = True)[0]
23         if confidence > 100:
24            confidence = 100
25         contents.append({"Name": name, "Confidence": confidence})
26     return {"statusCode": 200, "headers": {"Content-Type": "application/
           json"},"body": {'Labels': contents}}
```

extracts features from the input image. The multiway classification and box regression techniques are then used to obtain initial detection results. Finally, a non-maximum suppression is used to eliminate redundant results [57, 84]. To the purpose and scope of the evaluation presented in this Section, we have employed a well-known SSD trained model (*MobileNetSSD*) capable of identifying up to twenty classes of objects, including buildings. It is worth mentioning that a precise detection of more specific objects such as monuments, statues and other tourist attractions demands the proper training of a large image dataset containing these objects. In many cases, the training set must be annotated to guide the training process and thereby produce a more accurate model [84].

Implementation

The object detection pipeline was implemented as a function in Python language. In particular, Python has become a broadly used language among computer-vision practitioners and researchers, with several libraries offering support in Python. Among these, OpenCV —a popular computer-vision library implemented in C++ language— provides an API in Python. The same library also supports different deep learning methods.

Listing 6.1 presents the source code of the object detection function compatible with OpenWhisk platform. The function instantiates a *neural_network* object from OpenCV's deep neural network module (`dnn`). More specifically, it consists of an implementation of the *Convolutional Architecture for Fast Feature Embedding* (CAFFE) deep neural network framework. This complex object is then assigned to the global `cache` variable. In contrast with OpenWhisk native programming model, in which all objects composing a function are instantiated at each execution, this function exploits the proposed *in-memory cache* (see Section 3.3.2), which provides a significant advantage in terms of function initialisation overhead. The remainder of the function takes care of processing the image received as an argument and passing it through the neural network. Finally, the function returns all the objects detected in the captured scene along with their corresponding confidence probability.

6.2 Experimental Evaluation

In this section, we report on the experimental evaluation of the serverless architecture. The primary goal is to provide a quantitative assessment regarding *key performance indicators* such as latency, scalability, and elasticity. In particular, we followed the *Goal-Question-Metric* (GQM) methodology [15] to guide the evaluation process.

6.2.1 Goal-Question-Metric

Table 6.1 shows the questions and metrics for the experimental evaluation.

Q1: Resource-Constrained Edge Nodes

First and foremost, we evaluated the feasibility of deploying the components implementing a serverless architecture on resource-constrained edge nodes. In particular, we measured the memory footprint (**Q1-M1**) of all components in the idle state.

Table 6.1: *Goals, Questions, and Metrics of the Experimental Evaluation*

Evaluation Goal: The Serverless MEC Architecture		
#	**Question**	**Metrics**
Q1	*Is the proposed architecture suitable for resource-constrained edge nodes?*	**M1:** Memory footprint
Q2	*Is the proposed architecture suitable for latency-sensitive and data-intensive applications?*	**M1:** Overhead
		M2: Response time
Q3	*Is the proposed architecture scalable?*	**M1:** Simultaneous users
		M2: Function entropy
Q4	*Is the proposed architecture elastic (able to cope with abrupt workload fluctuations)?*	**M1:** Provisioning time

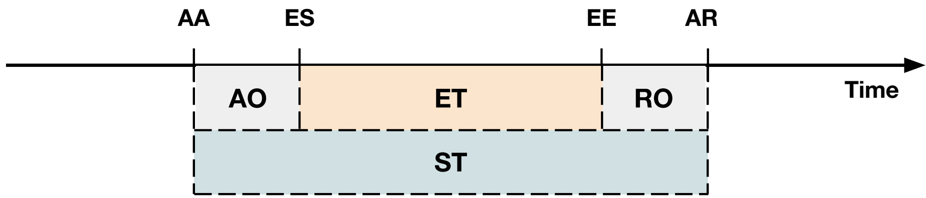


Figure 6.2: *Decomposition of the platform service time from activation arrival (AA) to activation return (AR), passing by execution start (ES) and execution end (EE). In additional to execution time (ET), the service time (ST) comprises the activation overhead (AO) and return overhead (RO).*

Additional evidence on the feasibility of deploying the serverless platform to the edge is provided with the experiments targeting the platform performance and scalability (**Q2** and **Q3**, respectively).

Q2: Latency-Sensitive and Data-Intensive Applications

Next, we evaluated the ability of OpenWhisk in satisfying the requirements from latency-sensitive and data-intensive applications.We focused on three metrics: the platform *overhead* (**Q2-M1**), defined as the difference between the total *service time* and the function *execution time*; and the platform *response time* (**Q2-M2**), which comprises both service time and network latency.

Compared with typical IaaS deployments, a FaaS platform adds processing and communication overhead while parsing events, managing the life-cycle of containers, and finally executing functions. Thus, the motiva-

tion for investigating the overhead of the FaaS platform to determine the impact of having additional components intermediating the interaction between latency-sensitive and data-intensive client applications and the runtime environment in which offloaded functions are executed.

In our experiments, the platform overhead is refined into sub-components, each of which refers to a specific part of the critical path from the function activation to its execution. While doing so, we considered OpenWhisk's public documentation and source code [104]. Despite the particular design and implementation choices, the rationale behind the operation of this mature FaaS platform is common to other open source solutions [106]. Platform overhead is decomposed in the following parts (see Figure 6.2):

- **Overhead:** the time taken by the platform to process an activation event and trigger the function execution. It also comprises the time between execution completion and sending the response in the case of *blocking requests*, which is the main scenario with computation offloading. It corresponds to the first metric in Question 2 (**Q2-M1**).
  - **AO:** the time elapsed between the function activation event and its execution within a warm container. Includes eventual initialisation overhead of the container and function (see Section 3.3.2).
  - **RO:** the time elapsed between the end of the function execution and the availability of results —fetched asynchronously or sent back to the source of a synchronous (blocking) invocation.
- **ET:** the execution time corresponds to the duration of the function execution. In our experiments, this is either a controlled variable (test function) or the actual execution time of a function implementing real application logic.
- **ST:** the service time comprises both overhead and execution time. It corresponds to the first metric in Question 2 (**Q2-M2**).

Q3: Scalability

In order to access the scalability of the serverless architecture, we considered the ability of the FaaS platform to accommodate larger and varied workloads without disruption of its performance. More specifically, we considered two metrics: *simultaneous users* (**Q3-M1**) and *function entropy* (**Q3-M2**). Both metrics were evaluated against distinct memory capacity configurations.

The first metric measures the impact of an increasing number of simultaneous clients on response time; it is directly related to the platform throughput for a given capacity configuration. The second metric measures the platform performance when subject to invocations an increasing number of different functions (hence, *function entropy*).

The user limit is a standard metric in software scalability testing. In contrast, the second metric is intrinsically related to the FaaS execution model. In contrast with a typical IaaS deployment, in which capacity defines a hard limit for how many concurrent services may be deployed, the fast allocation of containers enables the FaaS platform to rotate the current allocation in case of resource contention and thereby serve more functionality than a typical IaaS deployment. Such operation, however, has a non-negligible impact on performance, which we aim to evaluate.

Q4: Elasticity

Last but not least, the elasticity of the proposed architecture was evaluated in scenarios of sudden workload fluctuations. More specifically, we evaluated the platform scaling responsiveness (provisioning time, metric **Q4-M1**) when subject to bursts of workload in the following scenarios: when infrastructure is cold, i.e., no containers are allocated; and when capacity is thoroughly used by other functions, whose containers must be first deallocated.

6.2.2 Experimental Setup

The experiments presented in this section were performed with two dedicated Ubuntu/Linux 14.04 servers. Each server was equipped with 12 processors and 16GB of memory. Servers were connected to the same network through Ethernet at two hops of distance (separated by a switcher). The first server was exclusively used to host the FaaS platform stack. In turn, the second server was exclusively used to generate workload towards the first server.

As previously discussed, we adopted OpenWhisk, the most mature open source serverless framework available. OpenWhisk's development repositories receive daily commits. For our experiments, we used the last stable version at the time and followed the recommended set up using Docker Composer in a single machine. While the official documentation comprises distributed set up using Kubernets, we opted for focusing our evaluation on a single node, from which results can be extrapolated. The use of a

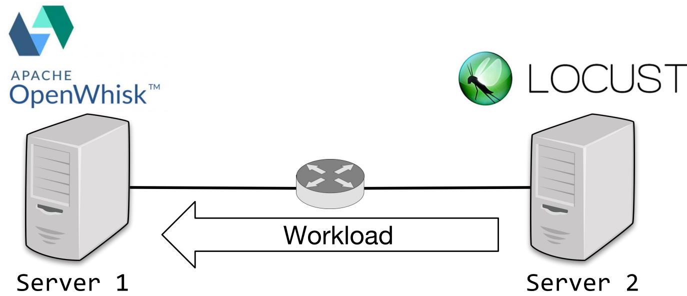


Figure 6.3: *Experimental setup. Server 1 hosts OpenWhisk, whereas Server 2 generates the workload according to the parameters in each experiment.*

single machine is also aligned with the idea that edge nodes are resource-constrained.

The memory capacity assigned to the single Invoker component in our setup was fixed for all but the experiments targeting the platform scalability. More precisely, we considered the memory capacity needed by the function been tested (e.g. 128MB for our Python 2.7 test function) and set the Invoker capacity to be 12 times the required memory. We found this number to be appropriate given the 12 CPU cores of the hosting machine.

The workload was generated from the second machine using Locust [1], an open source load testing tool. Locust provides a comprehensive API for Python language, based on which we implemented the test cases for each experiment and load scenario. We shall provide the details on the parameters used to generate workload for the evaluated metrics in Table 6.1 at the corresponding experiment section.

Except for the memory footprint, the experiments reported in this section were repeated ten times. For each of these experiments, thin vertical bars depict the standard deviation that corresponds to the plotted average.

6.2.3 Results: Memory Footprint

To evaluate the feasibility of deploying the platform onto resource-constrained MEC nodes, we measured the memory consumption of all the components that belong to the platform. More precisely, the values reported in Table 6.2 correspond to the memory footprint of each component while the platform is idle, that is, while no functions are being executed.

The total memory footprint, when the platform is idle, sums $1.8GB$. Additionally, each container instance (not depicted in Table 6.2) consumes

[1] https://locust.io/

Table 6.2: *Memory footprint of the serverless platform components (idle state).*

Fig. 8	Component	Memory Footprint (MB)
A	API Gateway/Nginx — Third-party	60 MB
B	Controller — OpenWhisk	520 MB
C	Kafka — Third-party	545.0 MB
D	FaaS Invoker — OpenWhisk	400 MB
F	CounchDB — Third-party	120.0 MB
G	Minio — Third-party	20 MB
H	Mosquitto — Third-party	9 MB
I	WS Proxy + Feed — Implemented	2x65 MB
	Total	$\approx 1.8\,GB$

a minimum of $128MB$ and a maximum of $512MB$ in its default configuration. Given the variety of deployments of MEC nodes, these results indicate the feasibility of deploying a complete Serverless MEC Platform onto MEC nodes with the resources of a personal computer. *Kafka* [51] was the most expensive component. While the role of a reliable messaging system is paramount for larger scale and distributed deployments such as regional-level MEC nodes with multiple invokers, resource-constrained MEC nodes would benefit from the combination of controller and invoker into a monolithic component.

Interestingly, OpenWhisk provides such a lean implementation. In this set-up, an invoker exists within the controller, while an in-memory queue replaces Kafka. This alternative set-up makes the Serverless MEC Architecture suitable for MEC nodes with constrained capacity.

6.2.4 Results: Overhead and Response Time

Platform Overhead

The FaaS platform overhead was measured with serial invocations to a single test function ($ET = 0$) in our first set-up deployment. In order to capture the possible impact of the allocation strategy used by this FaaS platform, we measured the overhead against an increasing inter-arrival time —to be more consistent with the architecture under investigation, from now on we shall refer to this variable as the *time between invocations* (TBI).

Figure 6.4 shows the obtained results for the overhead in two different *pause grace* (PG) configurations. The first (red bars) refer to the native OpenWhisk configuration, in which warm containers are paused after

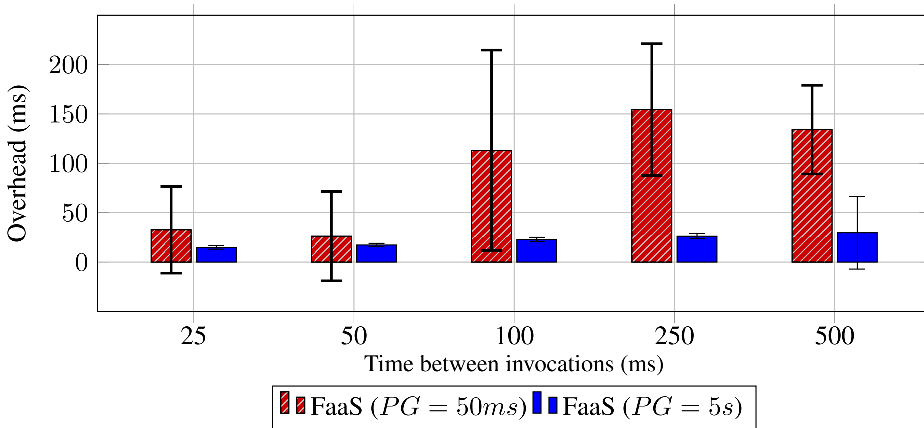


Figure 6.4: *Overhead vs Time between invocations. Each pair of bars depict the total overhead for a pause grace configuration of $50ms$ (red bar) and $5s$ (blue bar).*

$50ms$ interval of idleness. The second (blue bars) correspond to a higher pause grace value ($5s$). Obtained results show that the pausing (and resuming) of containers by the FaaS platform imposes considerable overhead whenever $TBI > PG$. The measured overhead in the native configuration ($PG = 50ms$) jumps from $\approx 30ms$ to $\approx 110ms$ when TBI goes from $50ms$ to $100ms$. In contrast, with our customised configuration ($PG = 5s$) the measured overhead remained stable at different TBA values —all of which bellow pause grace; in the worse case ($TBA = 500$), the complete overhead for executing the single test function was $30 \pm 37ms$.

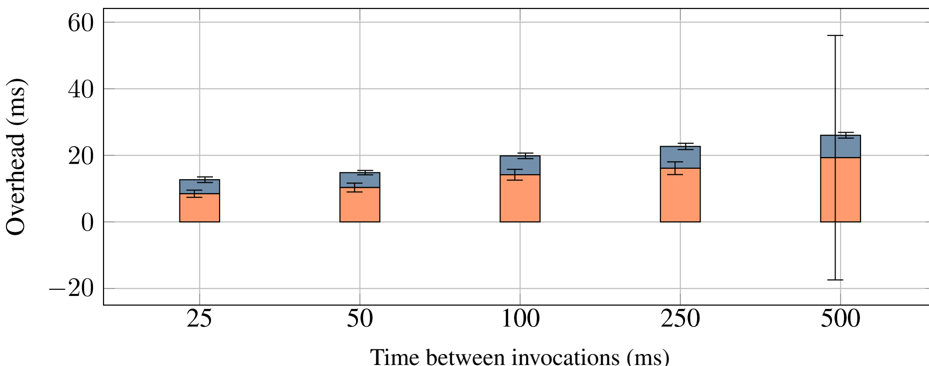


Figure 6.5: *Overhead vs Time between invocations. Each stacked bar depicts the overhead in terms of AO (bottom) and RO (top).*

Figure 6.5 refines the results obtained for the platform overhead ($PG = 5s$) in terms of *activation overhead* (in orange) and *return overhead* (in

gray). As expected, the activation overhead is more significant than the return overhead due to the time needed to process the activation event and dispatch it to a warm container. In the worse case ($TBI = 500ms$), activation overhead was $19.3 \pm 36.7ms$ —a substantial increase in the measured standard variation compared to the previous value ($1.9ms$). In turn, the corresponding return overhead was $6.74 \pm 0.86ms$.

In contrast with the experiments targeting the platform scalability, in which throughput was measured against a varying number of simultaneous users and disparate functions, we also measured the throughput of the serverless platform for a unique test function ($ET = 0$) consumed at an increasing rate. The goal is to evaluate how the platform performs in the advent of invocations to a single latency-sensitive function and thereby check whether it is capable of processing, given its overhead, a regular but intensive workload.

Figure 6.6 shows the obtained results for the platform throughput in terms of activations per second (A/s).In contrast with the results in Figure 6.4, in which points in the curve refer to the TBI, here the horizontal axis refers to the rate of invocations (thus, the inverse of TBI). Each experiment was performed for a fixed period of $60s$. Note that, in this and the remainder of the experiments in this section, the pause grace parameter in OpenWhisk was set to $5s$ (custom configuration).

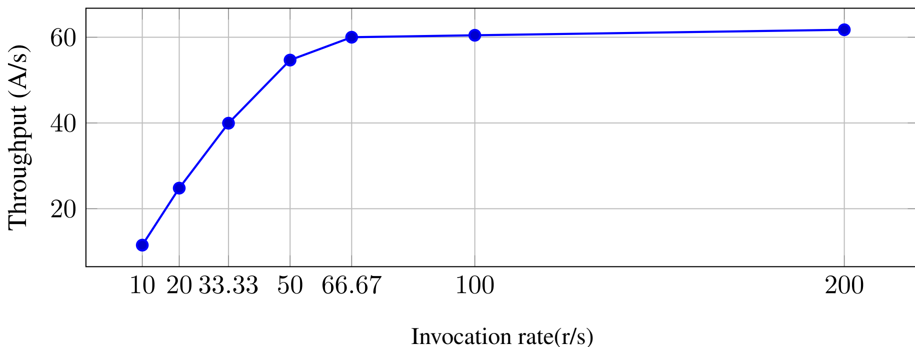


Figure 6.6: *Throughout vs Invocation rate.*

Obtained results show a steady throughput performance for up to 70 activations per second. From this point on, the measured throughput remains below the pace of invocations. At first glance, these results seem to indicate a low platform performance. However, as the experiments with multiple users in this section demonstrate, the bottleneck was actually on the client side: the single client's thread was not able to generate a higher

workload.

While the response time perceived by client applications still depends on both execution time and network latency, obtained results for the platform overhead corroborate the feasibility of the serverless architecture in satisfying the requirements of latency-sensitive applications.

Response Time

After assessing the overhead and throughput with a test function, we moved our attention to the performance of the serverless platform in executing functions implementing real application logic. More precisely, we measured the *response time* perceived by client applications that rely on the serverless platform for the execution of two functions, all implemented in Python 2.7: *object detection* and *face detection*.

The first function uses the SSD deep-learning method for detecting objects in an image (see Section 6.1.2). The second function uses a consolidated machine learning classifier (namely, *HAAR Cascade*) for detecting faces in an image. These computation-intensive, data-intensive, and latency-sensitive functions were selected for their relevance with respect to significant MEC use cases [41, 103].

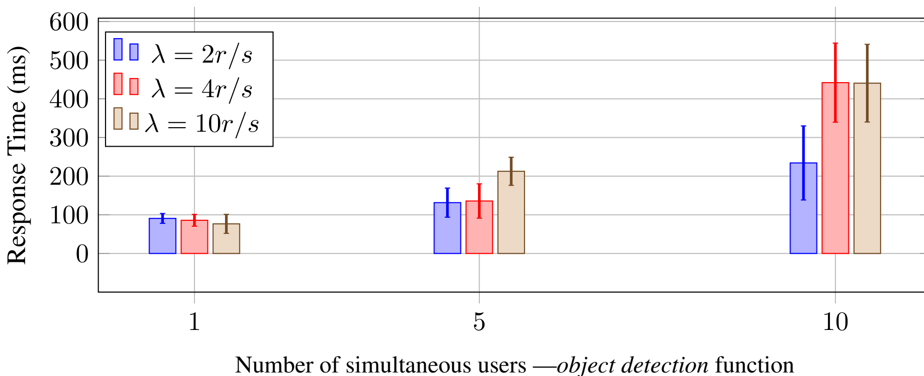


Figure 6.7: *Response time vs Number of simultaneous users*

We measured response time for each function separately with concurring invocations coming from an increasing number of clients for a fixed period of $60s$. Each client generates a synthetic workload using an exponential distribution set with an increasing average rate λ, where $\lambda = TBA^{-1}$.

Figure 6.7 shows the results for object detection. With an average invocation rate of $2r/s$, the highest response time was $210ms \pm 95ms$ for 10 simultaneous clients and half of that for up to 5 clients. For an aver-

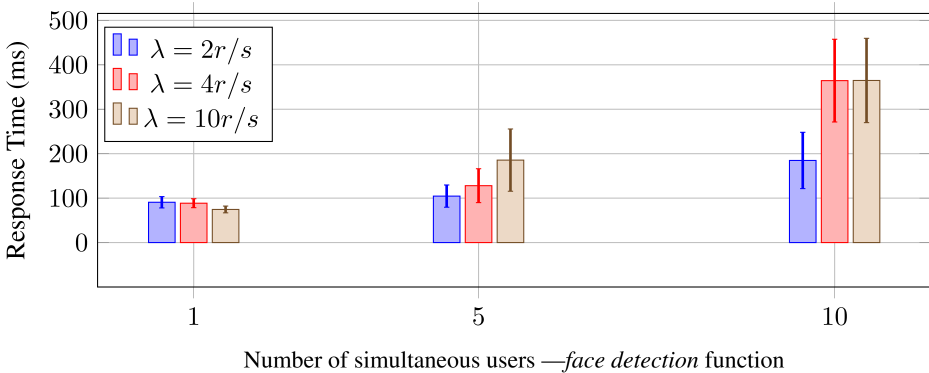


Figure 6.8: *Response time vs Number of simultaneous users*

age invocation rate of $4r/s$, response time increased linearly for up to 5 simultaneous clients ($136 \pm 44ms$) and jumps to $440ms \pm 100ms$ with 10 simultaneous clients —indicating the saturation of the platform. Lastly, at an average invocation rate of 10 r/s, response time was up to 56% higher ($441 \pm 102ms$) compared to lower invocation rates with 5 simultaneous clients.

Interestingly, the response time for an average rate of $4r/s$ and $10r/s$ was almost equal for 10 simultaneous clients. This unexpected result may be justified by a bottleneck in generating requests. Indeed, the results for the activation throughput in Figure 6.6 corroborates this hypothesis. Although this could be circumvented adding new machines to behave as clients, the substantial increase in response time from 5 to 10 clients at $4r/s$ was able to demonstrate the point in which the platform at the present set-up becomes overloaded.

The obtained results for the face detection function (see Figure 6.8) were slightly better (i.e., lower) in all scenarios. The measured response time was below $75 \pm 8ms$ for all invocation rates triggered by a single client; up to $186 \pm 70ms$ for 5 simultaneous clients; and up to $365 \pm 95ms$ for 10 simultaneous clients and average invocation rate of 10 r/s.

6.2.5 Results: Simultaneous Users and Function Entropy

Simultaneous users

To assess the scalability of the serverless architecture, we measured the throughput of the FaaS platform with concurring invocations to a test function coming from an increasing number of clients. In contrast with previous experiments, the test function hangs for a fixed period of $ET = 50ms$ to

emulate the execution time of a low latency function.

At each client, a synthetic workload was generated for a period of $60s$ using an exponential distribution set with an increasing average rate of λ. Once we have established the limits for a given deployment configuration, we repeated the experiments adding more capacity to the serverless platform.

Figures 6.9 to 6.11 show the obtained results for different capacity configurations. Each stacked bar depicts the throughput in activations per second (A/s) along with the corresponding response time, which allows us to visualise the point in which a given capacity configuration becomes overloaded. With the minimum capacity set-up ($4GB$), the platform was able to process as high as $526A/s$ with 60 simultaneous users and average invocation rate of $10r/s$. At this level, the perceived response time (i.e., round-trip time) was $112.6 \pm 25.6ms$, hence $\approx 62ms$ of overhead.

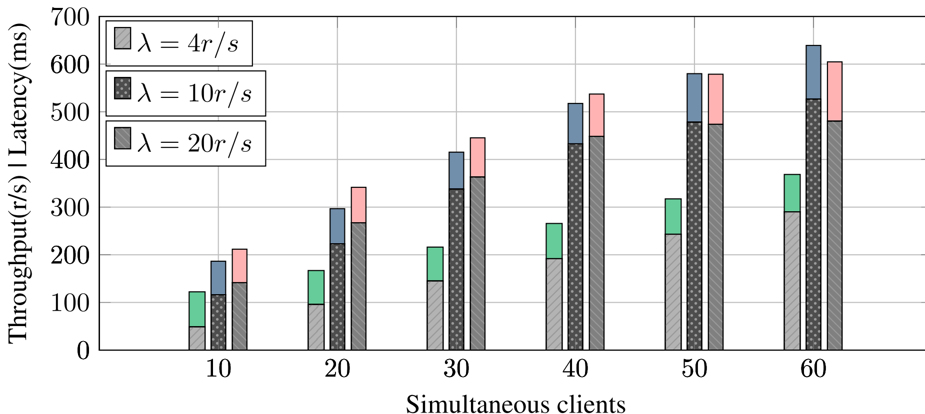


Figure 6.9: *Throughput | Latency vs Simultaneous clients — 4GB memory capacity.*

Results for $6GB$ and $9GB$ capacity configurations showed no significant improvement concerning the previous results. With $6GB$ of memory available, the platform throughput reaches as high as $529A/s$ with $\lambda = 20r/s$ and 60 simultaneous users, whereas the $9GB$ capacity set-up achieved as high as $517.563A/s$ in the same workload scenario. In both cases, the measured response time was $\approx 115ms$.

While these results seem to indicate a scalability problem —as the performance was not improved by additional memory— a closer look into the FaaS platform and our experimental setup revealed that CPU contention and not memory was the bottleneck. In short, the FaaS platform calculates the maximum number of containers based on the ratio between memory

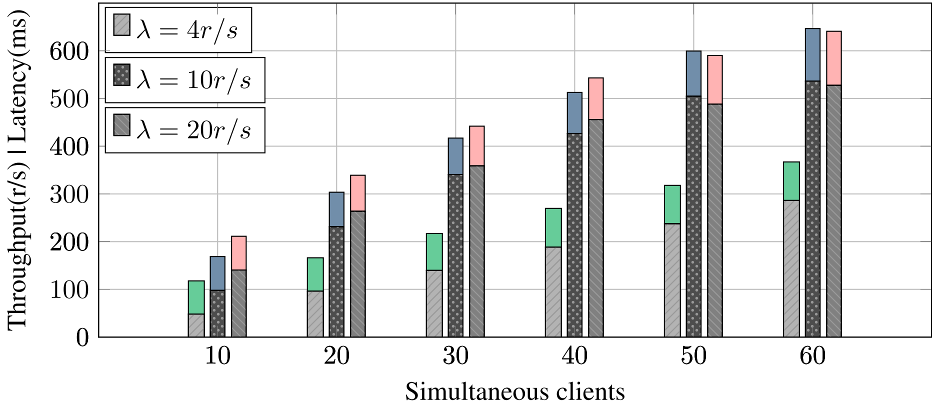


Figure 6.10: *Throughput | Latency vs Simultaneous clients — 6GB memory capacity.*

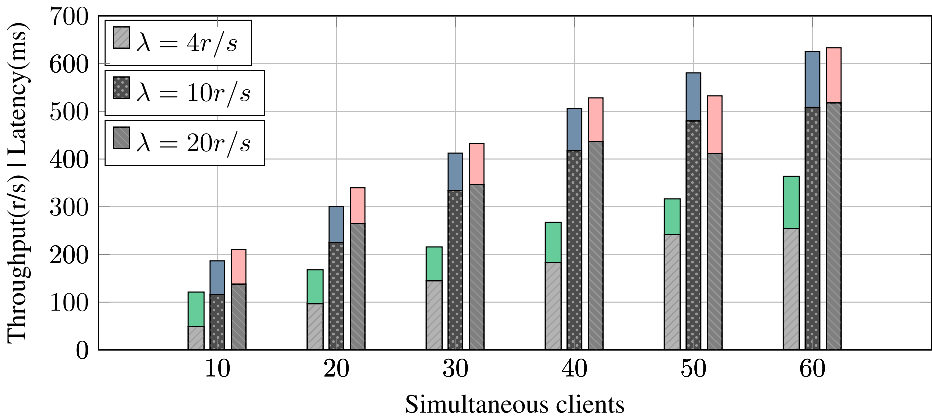


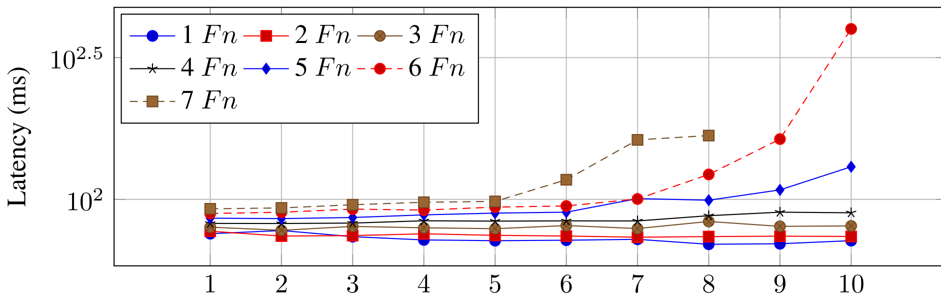
Figure 6.11: *Throughput | Latency vs Simultaneous clients — 9GB memory capacity.*

required by the function and the capacity available. Our test functions required $128MB$ only, which enabled the FaaS platform to scale up to 48 containers with the $6GB$ configuration and up to 72 with $9GB$ set-up. This outstanding elasticity, however, is not followed by the underlying processing capacity, as each container requires one core. Thus, as the number of containers (thus, processes) continues to increase, CPU contention prevents the platform from processing a more significant number of invocations.

Function entropy

As the last metric targeting the scalability of the serverless architecture, we measured the platform response time against an increasing number of different functions, after this referred to as the *function entropy*. In contrast to cloud-based IaaS deployments in which distinct components can be scaled horizontally in an independent manner, the OpenWhisk considers the maximum capacity allocated. In case of memory contention, containers that have been used less recently are released to free space for the processing of queued activations for which no warm containers are available.

The present experiment was performed following a different approach from the previous one targeting simultaneous users. Instead of increasing the number of simultaneous clients for a single test function and three different average invocation rates, we fixed the average invocation rate at a reasonable level ($\lambda = 4r/s$) and focused on the platform performance at different entropy levels and for an increasing number of simultaneous users. Not that, to simulate the case in which a single application relies on multiple functions, clients trigger a burst of requests for all functions at each entropy level. Similarly to the previous experiment, each function hanged for a fixed period of $ET = 50ms$ and required $128MB$ of memory.



Disparate functions invoked by 1–10 simultaneous clients with $\lambda = 250ms$.

Figure 6.12: *Latency vs Function entropy — 4GB memory capacity.*

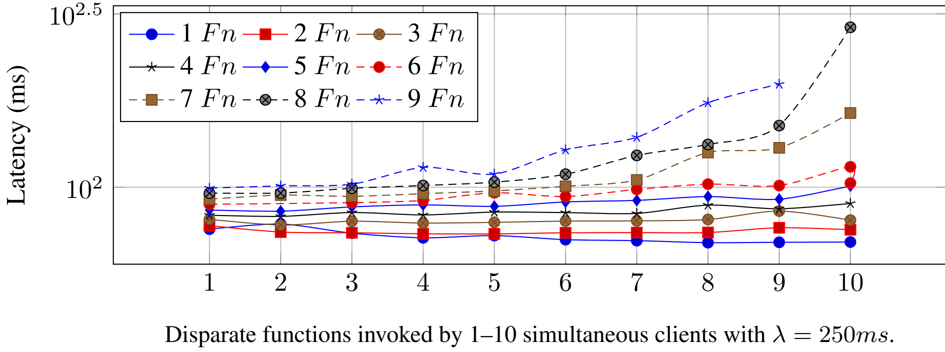


Figure 6.13: *Latency vs Function entropy — 6GB memory capacity.*

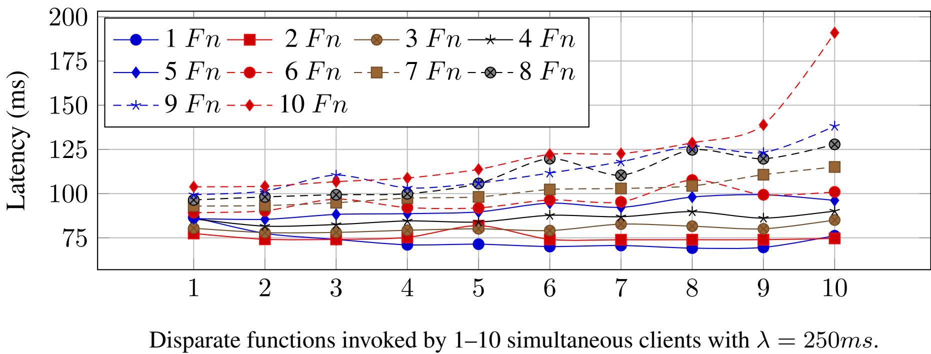


Figure 6.14: *Latency vs Function entropy — 9GB memory capacity.*

Figures 6.12 to 6.14 present the results for different capacity configurations. Points in the curves represent the response time for a given function entropy level and number of simultaneous users.

Obtained results for the more constrained capacity ($4GB$, Figure 6.12) show that the serverless platform handles up to 10 users concurrently triggering as much as 4 different functions in a burst without any damage to the perceived response time. With 5 different functions, the measured response time starts to increase with 9 simultaneous users. With higher levels of entropy, the performance hit is noticed earlier (with 7 and 6 simultaneous users, respectively).

In contrast, obtained results for $6GB$ memory capacity (Figure 6.13) show the ability of the serverless platform in serving up to 6 functions for 10 simultaneous users with negligible performance degradation, whereas 7 to 9 functions suffered from higher latency starting from 8, 7, and 6 simultaneous users respectively.

At last, the $9GB$ capacity configuration (Figure 6.14) exhibited substantial improvements. In this set-up, the serverless platform was able to process bursts of workload from up to 10 users concurrently triggering as much as 10 different functions. Performance hit was only perceived at entropy levels of 9 and 10 functions and for 10 and 9 users respectively.

6.2.6 Results: Elasticity

As the last set of experiments, we evaluated the elasticity of the serverless architecture by measuring its provisioning time in two different scenarios: when infrastructure is cold, i.e., no containers have been yet created; and when capacity if fully allocated, i.e., the FaaS platform needs to first release warm containers from its pool of resources before creating new ones.

The present experiments were also performed with a test function that hanged for a fixed period of $ET = 50ms$ and required $128MB$ of memory. At each scenario, a burst of requests was generated by 10 simultaneous users at an average rate of $\lambda = 4r/s$. The elapsed time was measured with a bash script loop; at each iteration, the container engine (i.e., Docker) was queried using the `Docker ps` command. The number of containers of the specific type was then logged along with timestamps.

Figure 6.15 depict the obtained results for the provisioning time of the FaaS platform. With the infrastructure cold, it took $\approx 0.35s$ to spawn the first container. Due to container reuse, the elapsed time for subsequent scale-outs varied. In total, it took $\approx 6s$ to scale from zero to a total of 10 containers (1 per simultaneous user).

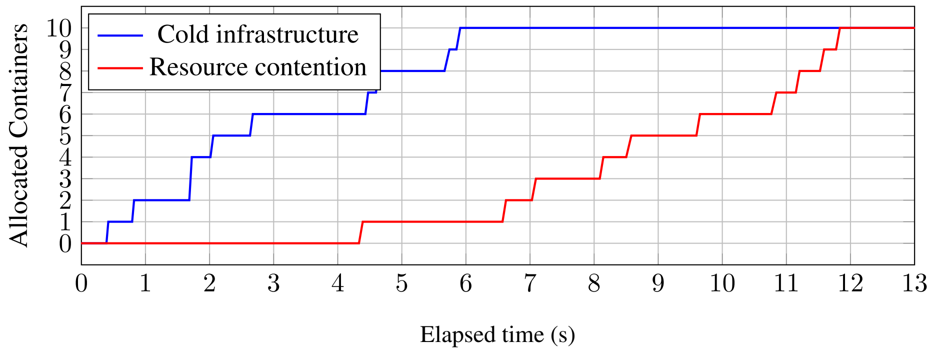


Figure 6.15: *Container provisioning time for cold infrastructure and resource contention scenarios.*

As expected, the obtained results for the scenario, in which capacity is fully utilised, show the impact of the container termination overhead on the provisioning responsiveness. In this scenario, the first container is created after $\approx 4.3s$, while the second shows up after $\approx 6.7s$. A total of $\approx 11.8s$ is elapsed after all 10 containers are finally available.

The previous results allows us to draw meaningful conclusions regarding the platform elasticity. OpenWhisk and the majority of container-based software nowadays rely on Docker and more specifically on *dockerd*. To the best of our knowledge, dockerd cannot create and terminate containers in parallel. Hence, even though the generated workload comprises parallel requests, containers are created sequentially. This behaviour explains the stair-like shape in the results for both scenarios in Figure 6.15.

The previous bottleneck limits the elasticity of the platform to some extent. Nonetheless, we argue that such an abrupt change of workload represents the worse possible scenario and that the few seconds needed to spin or terminate a container are acceptable. One must also take into account that, in a distributed deployment, the container orchestrator could leverage multiple hosts to spin containers in parallel.

6.2.7 Discussion

The obtained results show the importance of understanding the technology that implements a serverless architecture along with the configurations and strategies that suit the needs of a Serverless MEC Platform. In particular, we empathise the following aspects: (i) while the overhead associated with the creation and termination of containers is orders of magnitude lower than similar operations with virtual machines, it is still considerable for latency-

sensitive applications; (ii) the proper configuration of the FaaS platform in terms of *pause grace* is paramount for keeping this overhead low.

The experiments with functions implementing real application logic also indicate the feasibility of the serverless architecture in coping with the requirements from targeted use cases. For instance, if we consider interactive applications for whose frame rate must be no less than 5 frames per second to provide a seamless experience to its users (e.g. the MAR application in our Running Example), results for the object detection function are in-line with such requirements for up to 5 simultaneous users with a memory capacity configuration of $6GB$ only. Results show that, with additional capacity, the platform is able to scale to more intense workloads.

The proper dimensioning of the capacity is crucial for achieving the desired performance. Although evident from a theoretical point of view, the correlation between allocated capacity and performance of a FaaS platform is less evident due to its unique resource allocation mechanisms, namely the fast and reactive creation of containerised compute runtimes on demand. This is even more so in scenarios of resource contention, as the platform might need to terminate least used containers before spawning new containers needed to processed queued invocations. Thus, despite its flexibility and efficiency in providing more functionality with limited resources, a MEC operator must design the system according to: (i) the characteristic of the functions admitted into the system; (ii) the number of users.

Last but not least, it is also important to mention that our experiments were performed without GPU support. A MEC node equipped with the proper hardware acceleration could significantly reduce service time and elevate the achieved benchmark for a considerably higher number of users.

6.2.8 Threats to Validity

FaaS Platform

The first and more evident threat to validity is in the choice of OpenWhisk as the FaaS platform. OpenWhisk makes use of its orchestration technology to schedule function execution and scale containers. Contrary to other less sophisticated implementations, this particular FaaS platform uses a persistence component to log the metadata and results of function activations.

Many reasons substantiated the decision of using OpenWhisk as the FaaS platform of choice. First, it is the most mature open source solution available. Second, it is actively developed by the open source community and backed by a leading cloud vendor and other major IT companies. Third, the platform architecture is sound and well documented, which

allowed us to investigate in depth its behaviour and propose the modifications needed to cope with our targeted use cases and deployment scenario. It is also worth mentioning that some preliminary tests with another open source FaaS platform [106] revealed no significant gains that would justify its adoption in place of OpenWhisk.

Networking Technology

Another threat to the validity of our experiments regards the networking technology employed. Unfortunately, access to cellular infrastructure and wireless broadband technologies is minimal. One considered option was to use a standard wireless local area network (WiFi) to perform our experiments. However, this type of technology is yet to see its full potential in terms of throughput and jitter and therefore is not comparable to the performance to be achieved by 5G networks and accompanying Long-term evolution wireless broadband, a key technology enabler for MEC.

In light of this, we opted for generating the workload in our experiments through standard Ethernet technology, which provides superior throughput and stability. Another justification for this choice is in the infeasibility, given the resources at hand, to deploy a large number of client applications onto distinct devices. Instead, we opted for generating the workload from a single, but powerful server. Such configuration requires that the single link among these servers to have the necessary capacity.

Notwithstanding the importance of communication, our experiments focused on the platform rather than network performance. Throughout our experiments, the latency component introduced by the networking was always negligible. Hence, one must consider the additional network delay in a real deployment scenario when analysing the obtained results. In the other hand, we argue that high network performance is the basic premise of MEC and edge computing in general, which allows us to conclude that its effect on the overall performance should be minimal.

Object Identification

The object identification function used in our evaluation also deserves attention. In our experiments, we employed a popular trained model able to detect up to 20 classes of objects. This model does not include some of the POI that a MAR application for tourists would need to detect. Notwithstanding this, the employed trained model preserves two key aspects: large dimension, which prevents it to be packed as a function asset; and the same object identification method (Single Shot Detector), which assures accu-

racy and execution time to be comparable with an existing MAR approach in the literature [84].

6.3 A3-E Framework

We performed various experiments with four distinct domains to assess the Mobile-Edge-Continuum and the A3-E framework.

The first experiment studies how latency changes and how remote domains scale with a varying workload.

The second experiment evaluates all domains from a client's perspective, both in terms of battery consumption and total execution time.

The third experiment evaluates the capabilities of dynamically selecting domains and targets availability.

Finally, the last experiment evaluates the performance of A3-E's *Acquisition* and *Allocation*. When possible, we compared A3-E against Enorm, a state-of-the-art framework for edge node management.

6.3.1 Experimental Setup

Table 6.3 summarizes the four domains used in the evaluation. The mobile domain is part of our Mobile Middleware prototype (see Section 4.6.7), whereas the mobile application used for the experiments relies on an *object detection* function placed along the Continuum.

Our Domain Manager prototype (see Section 4.5) was deployed on two local-edge domains. *Local-edge-1* represented a situation in which latency is ultra low, but the computational resources are more constrained, and scaling-up is not possible due to the inherent physical restrictions of the underlying infrastructure (e.g., a lightweight, office-wide server). In contrast, *Local-edge-2* had more computational resources and, again, low latency could be achieved due to physical proximity (e.g., a robust edge server that is supposed to cover an entire floor of a building).

As for cloud domains, we used AWS Lambda [4] (*Cloud-FaaS*), the most mature FaaS solution on the market. To be consistent with our formulation of the Compute Continuum, functions and associated dependencies were deployed to the AWS Lambda data centre in Europe. Additionally, we also deployed the functionality onto a typical IaaS set-up (*Cloud-IaaS*) using virtual machines provided by the same vendor. The main goal of this set-up was not to compare traditional cloud services against a FaaS solution, but to demonstrate that the Compute Continuum could outperform the cloud under the tested circumstances and requirements.

Table 6.3: *Domains Setup in the Continuum for the Experimental Evaluation*

Domain	Machine Resources	Execution Environment
Mobile	Samsung Galaxy S6 SM-G90, 3Gb RAM, 8x Cortex CPU 2Ghz	Android 5.0.2 + Java Functions + OpenCV
Local-edge-1	ubuntu/trusty64-2, 4x vCPUs, 4Gb RAM	OpenWhisk, 256 Mb/Action, Python 2.7 + OpenCV
Local-edge-2	ubuntu/trusty64-2, 8x vCPUs, 16Gb RAM	OpenWhisk, 256 Mb/Action, Python 2.7 + OpenCV
Cloud-FaaS	N/A	AWS Lambda, 256 Mb/Function, Python 2.7 + OpenCV
Cloud-IaaS	Auto Scaling Group with t2.micro instances + Amazon Linux AMI 2017	NodeJs 6.11 server + Python 2.7 + OpenCV

6.3.2 Response Time and Scalability

The first experiment assessed different domains in term of response time and scalability when subjected to a varying workload. We simulated an increasing number of clients, each one making 100 requests for the functions required by the application at a rate of two per second. This setup was conceived by taking into account the default limit of concurrent executions in AWS Lambda [4] and Openwhisk [104].

This experiment excluded the mobile domain and focused on the remote domains, that is, the edge- and cloud-based ones. In this experiment, requests were emulated using Postman[2], an open source application designed to perform load testing. The payload used for this experiment was an example image of approximately $65KB$, which is a reasonable size for this use case when considering the requirements related to low latency and therefore fast computation time. To mitigate the cost of cloud-based providers, the execution time was profiled once for each domain, and then we focused on network latency, which is subject to higher fluctuations; the results on network latency were averaged through 5 executions.

Figure 6.16 shows the average latency for each increment in used clients. Note that the computation time (light grey) is different from the overhead (dark grey). The latter includes network communication (routing and forwarding) and queuing time (when no resources are available to process the request). If we use *Cloud-FaaS* as baseline, latency reduction was up to 90% for *Local-edge-1* and up to 82% for *Local-edge-2*.

Interestingly, with *Cloud-FaaS* the latency decreases when the number of simultaneous clients increases. This is due to the extra virtual machines

[2]`https://www.getpostman.com/`

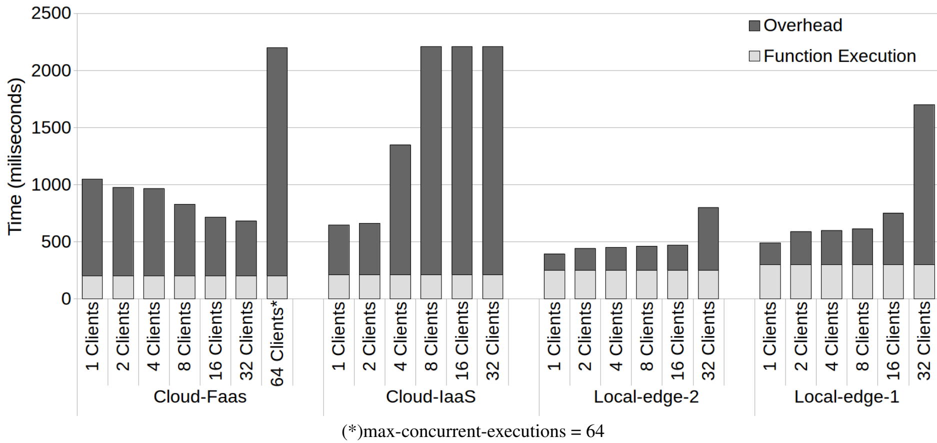


Figure 6.16: *Latency and scalability for each domain and increasing number of clients.*

provided under-the-hood by AWS Lambda to compensate for the initialisation overhead of cold requests: higher reuse rates correspond to higher stress levels of requests [58]. For a few service invocations, the load distribution adopted by AWS is uneven across hosts. This uneven use of the infrastructure may lead to the early deallocation of some containers (incurring in cold starts) if client workloads do not utilise all hosts. Thus, the decrease in the execution time with a higher level of workload is justified by the fact that more containers are kept warm.

Despite its virtually unlimited resources, the *Cloud-FaaS* has tunable limits for the maximum number of concurrent executions (default is 1000) due to budget constraints [117]. This is reflected in the substantial increase in latency when we consider the *Cloud-FaaS* domain and 64 clients.

If we use the *Cloud-IaaS* domain as baseline, the reductions when using *Cloud-FaaS* are up to 77% and 58% with respect to *Local-edge-1* and *Local-edge-2*, respectively. Interestingly, *Cloud-IaaS* outperformed *Cloud-FaaS* (46% less overhead) for light workloads (up to 2 simultaneous clients). This can be due to the additional steps performed by the API Gateway to forward RESTful calls to AWS lambda functions in *Cloud-FaaS*[3]. Nevertheless, this advantage is mitigated by the fact that *Cloud-FaaS* can better react to workload bursts due to its faster horizontal scaling [38, 117].

For light to medium workloads (up to 16 simultaneous clients), the overhead added by the local-edge domains is less than in both cloud alternatives. Under heavier workloads (from 32 simultaneous clients onwards) these domains present limited availability and degraded performance —as observed

[3] http://docs.aws.amazon.com/lambda/latest/dg/with-on-demand-https.html

with *Local-edge-1*, the most resource-constrained domain. One may argue that a local-edge domain is intended to cope with light workloads (e.g., a single office or a floor in a building). Edge domains that must cope with hundreds of simultaneous users (e.g., a mobile-edge domain), the computation and storage capabilities are expected to be orders of magnitude higher.

6.3.3 Battery Consumption and Execution Time

The main goal of this experiment was to evaluate the Compute Continuum from the client application viewpoint in terms of battery consumption and response time. Differently from the previous experiment, we set up a mobile device with our Mobile Middleware (see Sec. 4.6).

This experiment featured four different scenarios: the first three consider one domain each (*Cloud-FaaS*, *Local-edge-2*, and *Mobile-device*), and the last one (*all-domains*) combines the previous ones to form the Continuum[4]. The experiment consisted in cascading 2000 sequential requests for the *object detection* function passing an example image (with $65KB$ of size). We measured the total execution time, that is, the cumulated time between requests and their respective response; the battery consumption; and the average execution time per call. Figure 6.17 shows the obtained results, averaged on 5 executions for each scenario.

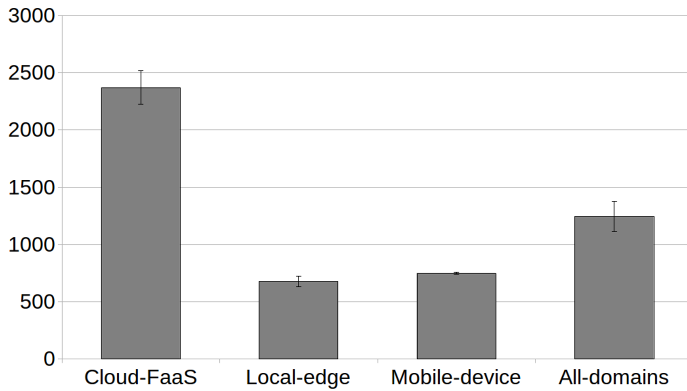
If we consider the total execution time (Figure 6.17a) and *Cloud-FaaS* as a baseline, *Local-edge* reduced it up to a 72%, while *Mobile-device* and *All-domains* up to a 69% and 49%, respectively. In turn, we measured the battery consumption (Figure 6.17b) in the *Mobile-device* domain, and noticed a drop of 4.5% after 750 seconds (i.e., 12.5 minutes) of execution.

Starting from the previous baseline, the savings with *Cloud-FaaS*, *Local-edge* and *All-domains* were 49%, 35%, and 49%, respectively. The time per call (Figure 6.17c), with *Cloud-FaaS* as baseline (1137 milliseconds per call), was improved by 76%, 68% and 47% for *Local-edge*, *Mobile-device* and *All-domains*, respectively.

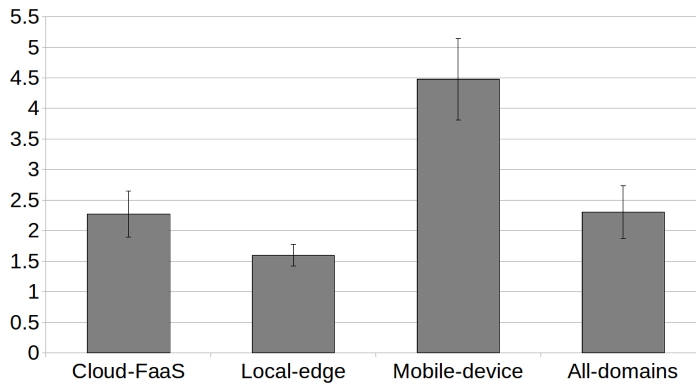
These experiments tell us that the total execution time when using only the cloud was two times higher than when using the Continuum (*All-domains*). Since the requests were performed in cascade and given the higher latency per call in the cloud, the total time increases accordingly. The use A3-E to switch to edge domains when possible would substantially reduce latency and would improve the perceived QoS.

Battery consumption was substantially lower when offloading computation, rather than performing it on the device. The *Mobile-device* domain

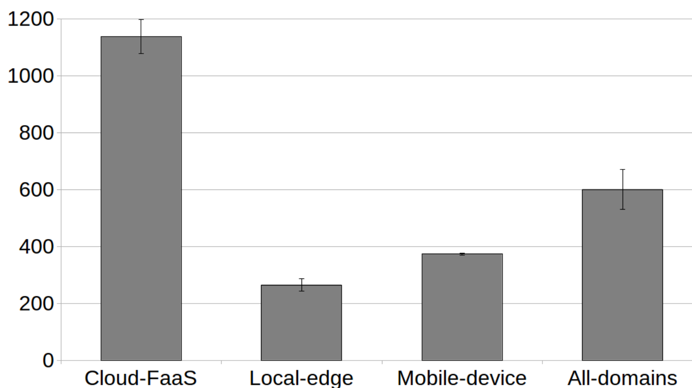
[4] Details on the availability of each domain in the *all-domains* scenario are discussed in Sec. 6.3.4.



(a) *Total execution time (sec)*



(b) *Battery consumption (%)*



(c) *Execution time per call (ms)*

Figure 6.17: *A3-E experimental evaluation results*

lasted half the time, but used twice as much battery (a prohibitive 20% of battery drain per hour) than with *All-domains*, given that it was performing CPU intensive operations at each request. This recalls the importance of computation offloading to preserve the resources of mobile devices.

6.3.4 Domain Selection and Availability

We evaluated the capability of A3-E to select the best domain, given the requirements on response time and battery consumption. In this experiment, we used the three domains above together to form the Continuum and simulated their availabilities using a probability distribution.

The mobile middleware was configured to ping for domain availability and network latency every two seconds. We considered that the cloud domain could be unavailable mainly due to the absence of mobile network coverage since the downtime of cloud services is minimal [32]. To simulate this, the average network unavailability was set to once every 15 minutes, while the average time for it to re-become available was 2 minutes, which resulted in an availability of 88%.

The rationale for the edge domain was analogous yet with a higher probability of being unavailable (e.g., due to the lack of memory or CPU). In this case, the edge domain was unavailable once every 10 minutes, and it needed an average of 5 minutes to re-become available, resulting in an availability of 66%. If we consider that edge nodes are only reachable within network coverage, the resulting availability is calculated by the product of the two availabilities, that is, $0.88 * 0.66 = 58\%$. Finally, the mobile device is considered as always available, as we aimed to stress the trade-off between battery consumption (when the mobile domain is employed) and latency (when remote domains are used).

To set an *optimal* baseline for this experiment, we calculated the theoretical number of calls to be served per domain, given the probabilities explained above. Upon this, we calculated the *optimal* execution by assuming no overhead for domain switching and by always using the best domain available.

Figure 6.18 shows the results for the *All-domains* scenario, with respect to availability. The experiment showed (see Figure 6.18a) an average perceived availability of 93% without considering the *Mobile-device* domain (5% and 35% improvement with respect to the cloud and edge domains, respectively), and 100% availability when also considering the *Mobile-device* domain (which is always available).

Figure 6.18b shows the distribution of calls per domain in the Con-

tinuum: 63% served by *Local-edge*, followed by *Cloud-FaaS* (30%) and *Mobile-device* (7%) —with an optimal of 66, 22 and 12% respectively. In terms of consistency, given that all requests were served, on average 70% of the requests perceived low latency, while the 30% that relied on the cloud had a degradation in QoS but still were processed successfully.

Finally, Figure 6.18c shows an increase of 33% in the total execution time using A3-E concerning the optimal, where the former includes the overhead of domain selection and switching.

The previous experiment showed that A3-E is capable of performing domain selection and offloading decision at runtime. The *All-domains* scenario reflects the underlying rationale of the Compute Continuum: one should exploit edge domains as much as possible by specifying appropriate weights in the QoS-requirements (see Equation 4.6), leading to a better balance among computation time, latency, and resource consumption.

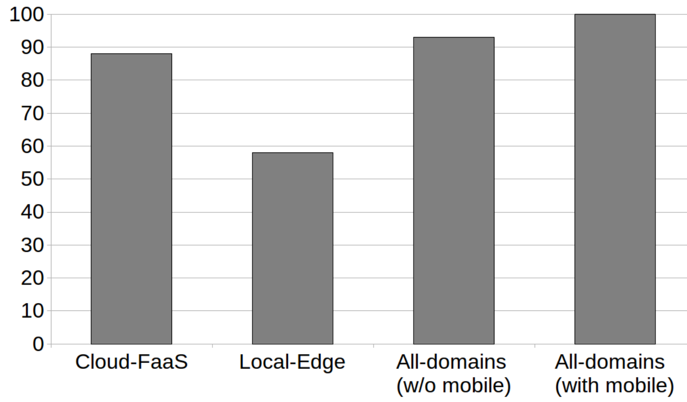
It is important to note that the response time of the mobile domain has the lowest possible score. Thus, the cloud domain was often selected as the first alternative in case of edge domain unavailability. It is also worth mentioning that, in our experiment, the mobile device was always available, hence the maximum availability when the mobile domain is included. However, in a more realistic scenario, the mobile domain may become unavailable in case of failures (e.g. involving a hardware or software component), resource contention (e.g. if there are multiple applications running in parallel), or even because specific components are temporarily disabled by users or the operating system (e.g. if battery level is too low).

6.3.5 Enorm

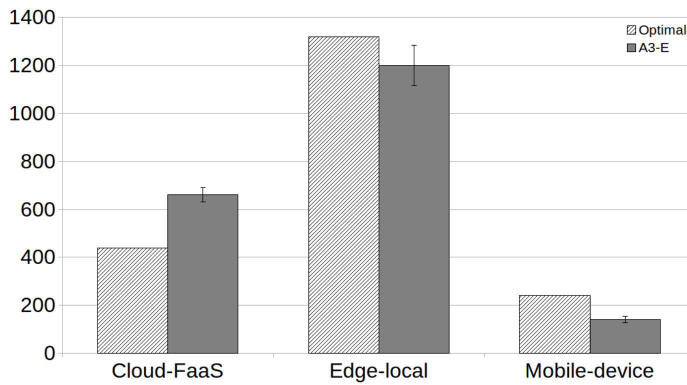
Finally, we compared *Local-edge-2* against Enorm [121], which assumes a similar, resource-constrained configuration for edge nodes.

Figure 6.19a shows the latency reduction when using both A3-E and Enorm, that is, an edge alternative, instead of a cloud-based solution. Both edge-based solutions are better than a cloud alternative for up to 50 simultaneous clients (from 35% to 55% latency reduction), and A3-E is always better (higher reduction) than Enorm. Both approaches perform worse than the cloud solution with heavier workloads (from some 50 to 64 simultaneous clients on).

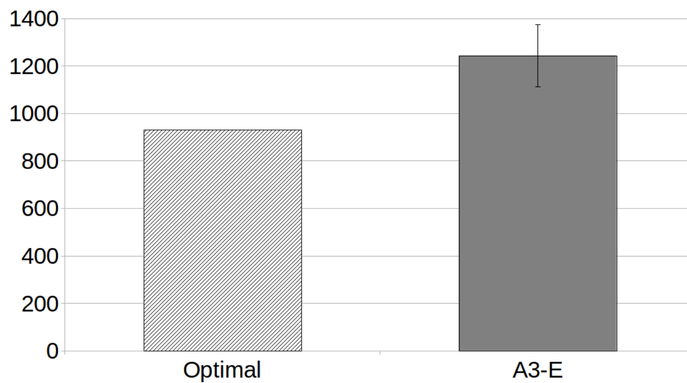
The last experiment evaluated the performance of *Acquisition* and *Allocation*. Given the resource limitations of edge domains and the potential benefits of the client arrival/exit awareness provided by edge locality, we targeted the evaluation of *Local-edge-2* as a domain.



(a) *Average availability (%)*



(b) *Number of requests served*



(c) *Total Execution Time (sec)*

Figure 6.18: *All-domains scenario results*

The experiment focused on the scenario in which no edge resources are pre-allocated, that is, a worst case cold-start scenario that includes both Acquisition (ΔAQ) and Allocation (ΔAL) overhead. The function we used was similar to the ones used in the previous experiments. Along with the metadata related to *client identification*, the client informed the server of the repository to fetch required assets (some $30MB$ including the function and dependencies). Also, the experiment only measured the domain-side performance, as the response time perceived by clients was evaluated in the previous experiments. After each successful Acquisition and Allocation, we uninstalled and deleted all the function-related assets to allow the following measurements to capture a worst case cold-start.

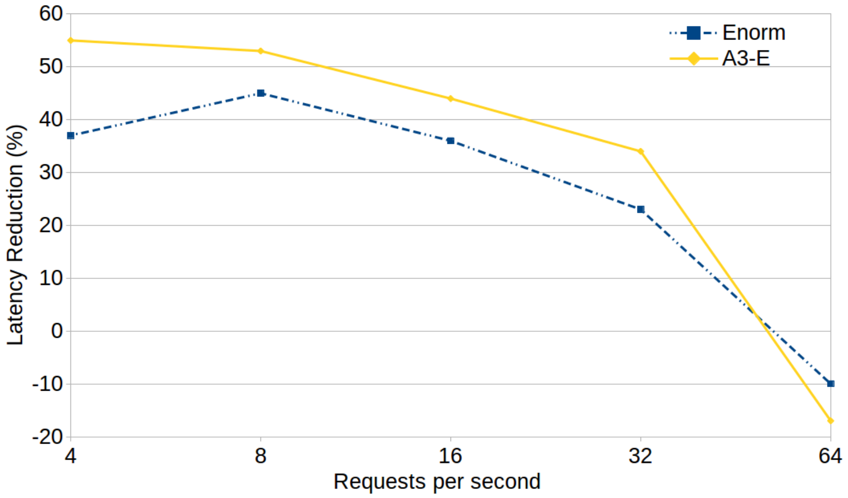
We executed cascading sequential requests for periods of 5 minutes, with different utilization levels of the edge node: low (10% server load and low network traffic, equivalent to 8 clients), medium (55% server load and 16 clients) and high (85% server load and 32 clients). Again, we were able to compare A3-E against the Enorm framework, given that our acquisition and awareness are analogous to the provisioning phase in Enorm, which consists of deploying application server partitions from the cloud to containers edge nodes.

Figure 6.19b shows obtained results. The average time between the detection of a *client identification* event and a successful Allocation was 12.5 and 44 seconds for A3-E and Enorm, respectively, without considerable variations regarding the current load of the edge node. Such a reduction of provisioning overhead (up to 70%) is one of the main advantages of adopting a FaaS model for the Continuum. The underlying FaaS solution (OpenWhisk in this experiment) reduces the burden of downloading and installing new functionality: thanks to the highly shared platform, functions can be created and deleted in a fraction of the time needed to do it with full containers (as in Enorm and most of the state-of-the-art solutions).

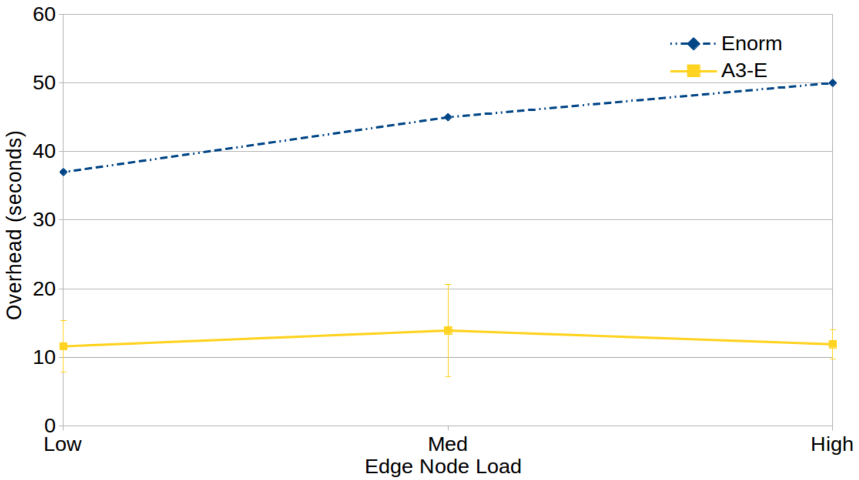
6.3.6 Threats to Validity

The experiments only targeted one example application and, in the case of the experiments of Sections 6.3.3 and 6.3.4, one single client device. Further tests are therefore needed to assess multiple clients that use several applications composed of functions with conflicting requirements, whose corresponding functions are deployed along the Continuum.

It is currently not possible to test with real mobile-edge domains, that is, to provide computational capabilities on base stations. This could be approximated either by simulation or by deploying edge domains by follow-



(a) *Latency Reduction (%)*



(b) *Provisioning Overhead (sec)*

Figure 6.19: *Comparison of A3-E (Local-edge-2) and Enorm against a cloud-based solution.*

ing the current specifications of MEC in terms of computational power and latency, to capture the heterogeneity of the Continuum better. Nonetheless, the fact that specifications and technologies are still under development limits the accuracy with which mobile-edge domains can be evaluated.

6.4 PAPS Framework

The simulator introduced in Section 5.5 was used to evaluate the allocation, placement, and scaling mechanisms of the PAPS framework given different partitioning of the MEC topology.

6.4.1 Experimental Setup

All the experiments were run using two servers running Ubuntu 16.04 equipped with processor Intel Xeon CPU E5-2430 for a total of 24 cores and 328GB of memory.

6.4.2 Partitioning

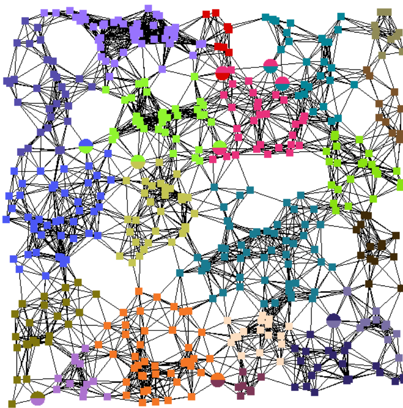


Figure 6.20: *Communities found in a large scale topology with 250 MEC nodes.*

Test	Conf	V	RT μ	RT σ	RT $95th$
OPT	10/50	6.4%	84.9	13.9	111.9
CT	10/50	0.6%	74.4	4.9	81.2
OPT	10/75	7.1%	89.6	15.1	113.4
CT	10/75	0.7%	75.6	7.8	81.8
OPT	10/100	8.9%	92.7	18.7	146.8
CT	10/100	0.9%	76.3	8.1	86.0
OPT	25/50	6.8%	92.6	16.7	176.0
CT	25/50	0.9%	75.6	10.7	85.8
OPT	25/75	10.5%	95.1	19.9	210.7
CT	25/75	1.8%	88.1	12.0	101.6
OPT	25/100	11.7%	101.6	23.0	221.3
CT	25/100	2.0%	85.8	20.0	107.3
OPT	50/50	7.4%	114.9	23.6	243.6
CT	50/50	1.4%	77.7	9.9	89.8
OPT	50/75	12.4%	118.9	27.6	260.6
CT	50/75	1.6%	78.7	15.6	91.6
OPT	50/100	14.0%	125.9	29.6	270.6
CT	50/100	2.2%	90.6	17.3	114.7

Table 6.4: *Obtained results.*

First, we assumed a large-scale edge topology of 250 nodes created with normally distributed node-to-node latencies. We used the SLPA algorithm to partition the topology in communities of 10, 25, and 50 nodes (parameter MCS) with membership probability $r = 0.35$. Fig. 6.20 shows the results of the partitioning when MCS was set 25. Coloured squares represent edge nodes within a single community; those belonging to overlapping communities are pictured as multi-colour circles.

It is worth mentioning that each *graph edge* in Fig. 6.20 represents a logical path whose network latency is within the inter-node delay threshold (e.g. $D_{MAX} \leq 10ms$). While node positions and edge dimensions may reflect the geographical location of corresponding MEC nodes, the visual representation is only illustrative and not necessarily proportional.

6.4.3 Allocation, Placement and Scaling

Next, we run two types of experiments to evaluate (i) the feasibility, the performance, and the scalability of the approach and (ii) the benefit of having a multi-layer self-management solution.

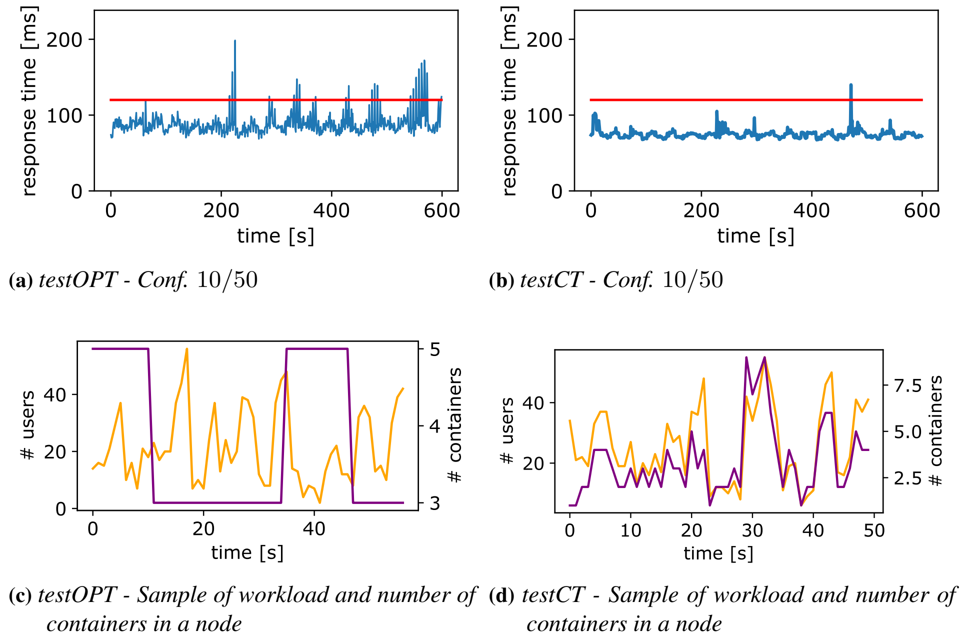
The first type of experiment, called *testOPT*, tests the community behaviour under an *extremely* fluctuating workload by using only the community-level allocation and placement. Each node keeps the target resource allocation to each running function constant between two community-level decisions. The second type of experiment, called *testCT*, uses both community-level and node-level adaptation to provide a more refined and dynamic resource allocation for the incoming random workload.

For each of the three community partitioning, we tested the system with an increasing number of types of functions: 50, 75, 100. Each execution lasted 10 minutes and tested one of the nine combinations of partitioning and number of functions. For each configuration, we executed 5 runs of $testOPT$ and 5 runs of $testCT$ for a total of 90 experiments.

The control periods of the community-level and node-level adaptation mechanisms were set to 1 minute and 5 seconds, respectively. If no feasible optimal solution is found at the community level, a constraint-relaxed version of the optimization problem introduced in Section 5.4.2 is solved, and the next placement decision starts 1 minute after. Moreover, we set the fraction of the marginal response time β to 0.5 and the value of the controller pole α (introduced in Section 5.4.3) to 0.9.

The workloads were generated using exponential distributions for both execution times (E_k) and inter-arrival rates. Specifically, the latter was generated for three different scenarios: low, regular, and high. A scenario was chosen randomly every 15 seconds to simulate an *extremely* dynamic traffic caused by user mobility and churn of devices. Finally, the RT_{SLA} of all the functions was set to $120ms$, whereas ET_{MAX} was set to $90ms$.

Table 6.4 shows the obtained aggregated results. Column $Test$ shows the type of test (either *testOPT* or *testCT*), column $Conf$ presents the configuration of the experiments (e.g., $10/50$ means that each community has 10 nodes and there are 50 different types of functions), column V shows



(a) *testOPT - Conf.* $10/50$

(b) *testCT - Conf.* $10/50$

(c) *testOPT - Sample of workload and number of containers in a node*

(d) *testCT - Sample of workload and number of containers in a node*

Figure 6.21: *Experimental evaluation results*

the percentage of control periods (5 seconds) in which the average response time violated the SLA, while columns μ, σ and $95th$ show, respectively, the overall average, the standard deviation and 95th percentile of the response time aggregated over the 5 experiment repetitions.

If we focus independently on *testOPT* and *testCT*, we can observe in Table 6.4 that even by increasing the number of nodes and functions the percentage of failures is kept under 14.0% and 2.2% respectively. These are reasonable values considering that we used an extremely variable workload with scenarios changing every 15 seconds.

Note that, the control period used for the community-level decision is four times larger than the time between two scenarios. Instead, if we compare the results of both tests, we can clearly notice the benefit of the node-level self-management; by enabling the control theoretical planners, the number of violations were reduced by an order of magnitude, for example, from 6.4% to 0.6% in configuration $10/50$, from 10.5% to 1.8% in configuration $25/75$ functions and from 14% to 2.2% using configuration $50/100$. Moreover, the average, the standard deviation and the 95th percentile of the response time are significantly lower in all the *testCT* experiments.

Charts in Fig. 6.21 help in better visualizing the results obtained. Fig-

ures 6.21a and 6.21b show the average response time of *testOPT* and *testCT* using configuration $10/50$. In the first chart, there are some violations (the horizontal straight line at $120ms$ is the SLA), while in the second graph there is just one at around second 500 and the response time is more constant (lower standard deviation) thanks to the faster actuation of the node-level manager.

Figures 6.21c and 6.21d show the number of users (lighter line) and the allocation (darker line) during the execution of a function on a single node for the two types of experiments (same configuration as before). In *testOPT*, the control period is kept larger, given the complexity of the optimization problem. Due to the highly dynamic workload, the allocation is often suboptimal and quite approximated from the actual needs of the tested system. On the other hand, during *testCT* the fast node-level container scaling allows the system to better fulfil the needs of the users by strictly following the course of the workload.

CHAPTER 7

Related Work

7.1 Serverless MEC Architecture

Ismail et al. [44] evaluated Docker, the leading container-based framework, as an edge computing platform. The paper discussed many of the benefits offered by container technology that are critical for edge computing. Their evaluation focused on four different criteria: (i) deployment and termination; (ii) resource & service management; (iii) fault tolerance; and (iv) caching. In their work, a test-bed was set up using a cloud database and three edge nodes interconnected by Ethernet network. They highlight three features of Docker that makes it an attractive technology for edge computing: fast deployment, small footprint, and excellent performance.

The benefits offered by Docker and the container technology highlighted by Ismail et al. are also leveraged by the Serverless MEC Architecture we propose. Indeed, many (if not all) the existing serverless offerings (e.g. [4, 43, 104, 106]) are based on Docker containers. Nonetheless, our proposal moves away from traditional user-centric service models —namely IaaS and CaaS— in favour of serverless computing to facilitate Operations, optimise the use of edge resources and boost the scalability of MEC.

EdgeScale [22] is another platform that leverages serverless comput-

ing to enable storage and processing on a hierarchy of data centres, positioned over the geographic span of a network between the user and traditional wide-area cloud providers. EdgeScale applications are structured as lightweight, stateless functions that can be rapidly instantiated on demand. In their poster presentation, the authors aim to implement all the functions, storage, routing and additional capabilities from scratch, while we opted for leveraging current open technologies such as Openwhisk. Besides, regarding the expected benefits of the approach, EdgeScale is on an early stage and does not report any empirical evaluation of concrete gains in terms of network latency, throughput and bandwidth.

Lambda@Edge[1] is an offering by AWS that allows one to explicitly deploy serverless functions to specific edge locations, closer to the user. Functions deployed to edge locations are not suited for general-purpose computing but for intermediating and optimising the interactions between edge servers hosting content and end-users. Also importantly, the notion of edge locations is coarse-grained: their edge schema, named CloudFront, consists of approximately 155 edge locations worldwide. In contrast, we consider densely distributed MEC nodes. If co-located with base stations, MEC nodes can be distributed every km^2 or less. Furthermore, the upcoming small 5G cells and microcells [16] allow us to think of one edge node per block, or even per building in certain vital places like government buildings, shopping centres and mass transport stations.

The first real-world prototype of the mobile edge (by Nokia Siemens and Intel [75]) features base stations equipped with commodity hardware, and application deployment is based on virtualisation and containerization technologies. The authors stress the benefits of the mobile edge architecture for both mobile service consumers and operators. Applications running on the mobile edge are expected to be event-driven, which is in-line with the Serverless MEC Architecture that we propose. However, this approach does not take into account the inherent resource limitations of edge computing, tackled by our architecture by adhering to the FaaS execution model. Besides, the authors present a taxonomy of MEC applications that can profit from MEC deployment. Interestingly, the MAR application —used as Running Example along with this thesis— is representative of two of the most benefited application scenarios, namely *offloading* and *augmentation*.

Satria et al. [93] proposes two different recovery schemes for overloaded or broken MEC nodes. One recovery scheme is where an overloaded MEC node offloads its workload to available neighbours within transfer range. The other is for situations in which no adjacent MEC node is within transfer

[1] http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html

range, and user devices are employed as ad-hoc relay nodes.In the former scheme, one MEC node performs the role of a cluster head. The latter provides offloading alternatives to the MEC nodes within the cluster, whereas cluster members inform the head about their current state (i.e., overloaded or not). The second scheme considers the signal-to-noise, buffer size, and bandwidth capacity of relay nodes. It then formulates an optimal data allocation problem, whose solution defines the ad-hoc routing from disconnected devices to a recovery MEC through selected relay nodes.

Similarly to the former approach, in our proposal we harness the collaboration among MEC nodes to boost the availability, resilience and scalability of the proposed Serverless MEC Architecture. In contrast with the leader-based scheme proposed by Satria et al., MEC nodes establish proactive recovery bounds in a fully decentralised manner. While the idea of employing user devices as an ad-hoc relay to extend the range of MEC node collaboration is attractive for operators, we argue that users may not be willing to have their resource-constrained devices used for such purposes. This is especially so if it incurs in battery drain due to additional use of radio communication. Thus, we opt for limiting collaboration among surrogate MEC nodes interconnected through high throughout, reliable network links.

7.2 A3-E Framework

Satyanarayana and his colleagues were the first to introduce the notion of a compute continuum with cloudlets —trusted, resource-rich computers, or clusters of computers, connected to the Internet and available for nearby mobile devices [96, 97].

Cloudlets can be seen as the first *Mobile-Edge-Cloud* architecture. Its goal is to reduce the response time perceived by interactive applications. The Mobile-Edge-Cloud Continuum extends the cloudlet architecture with other types of edge-centric deployment configurations —namely mobile-edge (MEC). Moreover, in our model cloud deployments are still regarded as a highly available alternative to mobile and edge computing.

Amongst other domains, the A3-E framework intermediates the opportunistic interactions between local-edge domains (cloudlets) and mobile applications (referred to by Satyanarayanan as *cyber foraging* [94]). Nonetheless, there are some significant differences with the original cloudlets proposal. First, cloudlets rely on dynamic VM synthesis rather than container technology [97]. This might be justified by the time in which the cloudlets architecture was proposed, which pre-dates the consolidation of the con-

tainer technology. On the other hand, even if containers were adopted, cloudlets remain very general regarding the type of components it can host or how they should be scaled to cope with dynamic workloads. Also importantly, rather than pre-defining if and where computation will be offloaded from mobile devices, A3-E enables this decision to be made at runtime. Thus, the A3-E framework address essential gaps in the materialization of the cloudlets vision and architecture.

Sarkar and his colleagues [92] compared the performance of an edge-cloud continuum (bridged by means of a fog computing tier) and traditional cloud deployments in the context of IoT. The authors demonstrated from a theoretical point of view that the edge-cloud continuum outperforms cloud computing as the fraction of latency-sensitive applications is 50% of all deployed applications. However, in scenarios where the majority of the applications do not have strict latency requirements, the continuum is observed to be an overhead compared to cloud-only execution.

Several works (e.g. [46,55,78,130,138]) tackled the decision of whether to offload computation from/to mobile, edge, or cloud nodes.

In this category, Orsini et al. [78, 79] proposed CloudAware, a comprehensive context-aware mobile middleware that handles offloading to edge and cloud nodes. CloudAware's primary goal is to provide mobile applications uninterrupted availability. The latter is achieved either by offloading computation to surrogate edge nodes, to cloud data centres, or executing tasks locally. More importantly, the proposed framework takes into account the conditions of both logical and physical environments, as well as other contextual information in the client-centred offloading decision.

CloudAware is based on Jadex, an agent-oriented middleware that ships with various tools and features such as service discovery and different communication protocols. It comprises four main components: a partitioner that determines which parts of the code are candidates for offloading; a context monitor that senses and analyses contextual information; a solver that decides whether to offload computation and to which node; and a coordinator that, among others, handles discovery and synchronization.

At the hearth of CloudAware, a Generic Context Adaptation (GDA) forecasts, out of a provided feature set, the availability of WiFi, the bandwidth range, and other attributes such as the execution time of an offloaded task. The GDA predicts the previous context information based on historical data collected by the mobile device. The authors relied on a mobile dataset collected from 20 users for a period of 18 months to obtain a trained model. Among other information, the training data contain the battery level, which was used to infer the battery drain for various tasks.

CloudAware was evaluated regarding multiple attributes. Similarly to A3-E, it maximises availability by including the mobile device as a fallback in case of unavailability of surrogate nodes. Likewise A3-E, the mobile middleware selects the alternative that best suits specified requirements and the context of operation.

There is also a similarity between the concepts of *tasks* (CloudAware) and serverless functions (A3-E). In both cases, the granularity level can vary according to the application design. In contrast with active components supported by Jadex, the paradigm of serverless computing and the FaaS model have received considerable attention in recent years. In terms of portability, functions can be executed in heterogeneous environments (including mobile devices) and written in different languages. Conversely, Jadex is limited to Java. On the other hand, Jadex provides out-of-the-box features like service discovery, which are implemented by our middleware using state-of-the-art protocols.

Although the authors provide a general description of CloudAware main components, little is said about their behaviour. For instance, a challenge regarding the monitoring of contextual data like battery level is the lack of precision. Some authors overcome this limitation through modifications to the hardware or operating system, which renders their approach not practical. A3-E makes use of information provided by the Android platform. Unlike CloudAware, we provide a detailed description of: how we obtain context information; how this information is used in the selection of the best domain for the execution of functions.

Last but not least, CloudAware uses a prediction technique to anticipate the context and thereby improve the success rate of task execution. While this is an interesting approach, we add some important considerations. First, multiple factors may undermine the prediction accuracy. Also, the training process is not trivial and, as stated by the authors, likely to be performed a priori by remote servers. Whenever the actual conditions diverge from the training dataset, the prediction error is likely to increase. Prediction errors may hurt the offloading decisions by preventing tasks from being offloaded even if surrogate nodes are available. In A3-E, offloading decisions are reactive regarding the monitored context. The rationale is as follows. Serverless functions offloaded to edge nodes are stateless and expected to be short-lived. While mobility may disrupt communication, the mobile device is expected to remain in proximity for at least enough time for several executions to be completed. Hence, the eventual error caused by mobility should not affect the normal application behaviour; instead of not trying to offload execution, fault tolerance may react

by rescheduling the request to the new surrogate node.

Regarding scalability, FaaS harnesses the container technology to achieve fast scaling, whereas the scalability of CloudAware depends on the underlying framework (Jadex). Our implementation of A3-E's mobile middleware also harnesses adaptive strategies to reduce the overhead of context monitoring and domain changes. With respect to security, serverless functions are executed in isolation at the software level (containers). In contrast, CloudAware depends on the compute runtime provided by Jadex. At its current version, our framework does not tackle the authentication of function requests. Nonetheless, it makes use of encrypted connection and delegates authentication to the FaaS platform.

Next, we discuss some state-of-the-art frameworks. In contrast with CloudAware [78, 79] and A3-E, the offloading decision is delegated to a remote server, which is fundamentally different from our proposal. Hence, we limit the discussion to the essential characteristics of each framework.

Cuervo et al. proposed MAUI [21], a context-aware offloading framework whose primary purpose is to minimise the energy consumption of mobile devices. MAUI leverages the properties of the Microsoft .NET Common Language Runtime (CLR) to achieve method-level offloading. Similarly to CloudAware, developers must annotate candidate methods. In contrast, A3-E leverages a language-agnostic App configuration file, which allows the specification of functions that are not executable locally (e.g. edge-only and remote-only).

On the mobile device, MAUI architecture consists of a client proxy, a profiler, and a solver. The client proxy handles control and data transfer for offloaded methods, whereas the profiler instruments the application and collects measurements of its energy and data transfer requirements. The MAUI server architecture comprises a profiler and a server proxy with similar roles performed by their client counterparts; the actual decision engine (solver) and a coordinator (controller). The solver component solves an Integer Linear Programming (ILP) problem to decide whether offloading can save energy or not. More specifically, this is performed on the MAUI server to prevent battery drain, while the client-side solver provides an interface to the decision engine. The MAUI Controller manages authentication and resource allocation for incoming requests. The server can run both on a public cloud or on private cloud deployments.

The optimisation problem is re-solved periodically to adapt to possible changes in network conditions. The formulation considers call graphs of method invocations. The objective function minimises the energy consumption sum, subject to latency constraints.

MAUI is able to estimate the energy consumed when executing a method locally through a linear model, built using *Least Squares Regression*. The model correlates CPU cycles to the measured energy consumption. Similar models are built for WiFi and 3G interfaces, but further details are omitted. The energy measurements have been obtained using a hardware power meter, connected directly to the smartphone's battery terminals and able to sample the current drawn with a frequency of 5000 Hz.

The MAUI framework tries to predict execution times using a history-based approach. However, it does not consider the input size, since it is assumed that it will not vary much between executions. It has been shown elsewhere [73,115] that such assumption may lead to imprecise predictions.

The limitation of the MAUI framework regarding real-time applications is twofold: (i) it targets the minimisation of battery consumption without considering response time; (ii) the scheduler (solver) is located on the MAUI server. Overhead is introduced before the mobile device is informed about the scheduler's decision. In contrast, A3-E privileges client-centric offloading decisions. The latter follow a more straightforward MCMD algorithm. On the other hand, it supports a broader set of requirements. More importantly, its low complexity allows resource-constrained mobile devices to computed it locally.

On the server side, A3-E benefits from the container technology and the FaaS model, whereas MIAU relies on statically allocated VMs to host their solution. Thus, A3-E outperforms MIAU in terms of scalability.

Kosta et al. proposed *ThinkAir* [49], a context-aware framework that supports method-level offloading to a smartphone runtime clone deployed to a cloud-based VM. ThinkAir was designed to improve the scalability of the MIAU framework, allowing clients to achieve their desired QoS either by executing on multiple cloned VMs in parallel or by asking for a more powerful VM clone. Although able to improve the scalability of the previous framework, ThinkAir still relies on typical IaaS deployments. Moreover, offloading decisions are still delegated to a server-side component.

Closely related to our infrastructure model, Tarneberg et al. [114] tackled the application placement across heterogeneous edge and cloud infrastructures. The authors stressed the challenges in satisfying application requirements while minimising infrastructure-wide operational expenditure. They also highlight the importance of load balancing to mitigate resource usage skewness. Their main contributions are twofold. First, they present a model that captures the heterogeneity of the decentralised infrastructure in terms of resources and cost; application requirements, and workload. Second, they propose an algorithm to solve the resulting multi-optimization

placement problem, which considers application execution cost in terms of computer and network resources, as well as the overload penalty on each node in terms of latency and resource utilisation. The objective function reflects mostly the interests of the infrastructure provider (e.g. a telecom operator), whose primary goal is to minimise the overall running cost.

While the edge-cloud continuum is part of our infrastructure model, our framework focus on the autonomic capabilities of individual domains (or data centres, using the authors' terminology [114]). A3-E tackles the deployment and scaling of stateless and lightweight functions —instead of monolithic applications— onto mobile, edge, and cloud platforms according to application requirements, self-managing policies and the availability of resources. As previously mentioned, centralised orchestration solutions may complement A3-E, as they could be employed by providers of multiple inter-connected domains to optimise their costs (e.g. by retaining or releasing resources at different mobile-edge domains in specific contexts of operation). In turn, A3-E adapts to the conditions imposed by the geo-distributed infrastructure operator (e.g. by deallocating low-priority functions and selecting alternative domains for the affected applications).

Zhao et al [138] proposed a cooperative scheduling in which compute tasks are offloaded either to local clouds (e.g. cloudlets) or internet cloud (i.e., typical cloud data centres). The local cloud is modelled as an M/M/C queue. The authors proposed a priority-based policy for scheduling tasks according to their tolerance to delay. A task is admitted by the local cloud if a server is available, or if the corresponding buffer is below a threshold. The authors then propose an algorithm for optimally determining the buffer threshold for different priorities. The performance of the priority-based scheduling was compared against a first-come, first served policy (FCFS), a greedy policy, and a non-buffer policy. Results show a 5% improvement in success rate compared to the FCFS policy, whereas the use of Internet cloud as fall-back improves the success rate in 20%.

A3-E also makes use of policies to solve contention among functions. Requests are served based on their arrival order and priority. In opposition to Zhao et al 's approach, A3-E emphasises client autonomy. An overloaded edge may deny the execution of a request and will not delegate its execution to the cloud. Given the perceived context of operation (e.g. denied request), the mobile middleware has the ultimate decision of where computation is performed (e.g. locally or the cloud). While our solution benefits from standard handover technique to preserve the connection with a cloud, a device connected to the local cloud scheduler may not be able to retrieve results if it moves to a base station served by another local cloud.

The ENORM framework [121] for edge node resource management shares similarities with A3-E. In ENORM, cloud managers are responsible for the deployment of server-side application partitions onto edge nodes with the aim of reducing service latency and the volume of data sent to the cloud. Server allocation follows a handshake between cloud and edge managers. If successful, application partitions are deployed to edge servers and clients are bound to specific ports following a network-level reconfiguration.

The ENORM framework architecture includes the following components on the edge node: the Resource Allocator keeps track of the available CPU cores and memory; the Edge Manager is composed by the node manager (deals with the requests obtained by the server manager from a cloud server) and server manager (initialises containers, allocates ports and security); the Monitor periodically tracks communication/computing latency of each application edge server; the Auto-scaler dynamically allocates/deallocates hardware resources to the containers executing application servers based on monitored metrics; finally, the Application Edge Server is the partition of the cloud server hosted on the edge node.

The ENORM framework is mainly focused on resource management from the cloud to the edge (top-down). In contrast, A3-E adopts a bottom-up approach in which autonomic edge domains and potential consumers of serverless functions interact directly: functions are dynamically deployed after a handshake between clients and autonomic edge domains; clients decide on the best domain given their requirements and perceived QoS. Depending on the requirements (e.g. latency-sensitive functions), edge nodes are likely to be first in the domain rank and then going up in the hierarchy given the infeasibility of running by or in proximity to the mobile device.

Like in ENORM, our framework takes into account both priority and latency to solve contention among functions. Nonetheless, in A3-E requests from different clients are not bound to specific instances, but decided by a load balancer, favouring seamless mobility and scalability. Accordingly, allocation and scaling in A3-E are carried by a node-level control able to deal with a highly dynamic workload. On the other hand, ENORM provides support for stateful applications, whereas in our model stateful components are hosted locally or provided by typical cloud deployments.

As for the Network Function Virtualization (NFV) field, several frameworks, mainly based on the ETSI MANO standard [45], have been proposed to cope with the fluctuating demand of network infrastructure[2] [107]. These frameworks provide an abstraction layer over a mobile edge infras-

[2]For example, OpenBaton: https://openbaton.github.io/

tructure, making the shift among the different parts of the continuum utterly transparent to applications. However, A3-E focuses on the opportunistic placement of application-level functions (following the FaaS model) rather than on VNF elements. We see NFV/VNF as a part of the underlying infrastructure, thus the work on them is complementary to ours.

7.3 PAPS Framework

A number of works tackled the problems of placement, scaling, and migration of services in geo-distributed infrastructures of various density levels. Some (e.g. [28, 114, 132, 136]) focus on the previously mentioned problems for a set of single-component applications, whereas others (e.g. [26, 71, 124]) consider multiple-components. In some cases (e.g. [26, 114, 135, 136]) the solution refers to singleton components, whereas others (e.g. [28, 71, 132]) tackle the placement of multiple component instances.

Nastic and his colleagues [72] introduced a serverless edge data analytics platform. Their platform extends the notion of serverless computing to the edge via a reference architecture, enabling uniform development and operation of data analytics functions. The proposed serverless data analytics paradigm is particularly suitable for managing different granularities of data analytics approaches bottom-up. This means that the edge focuses on local views (for example, per edge gateway), while the cloud supports global views, that is, combining and analysing data from different edge devices, regions, or even domains.

Data analytics can be performed on edge nodes, cloud nodes, or both, and delivered from any of the nodes directly to the application, based on the desired view. Moreover, the top-down control process allows decoupling of application requirements (the what) from the concrete realisation of those requirements (the how). This allows developers to define the analytics function behaviour and data-processing business logic and application goals instead of dealing with the complexity of different management, orchestration, and optimisation processes.

Within Nastic and al's proposal, the Serverless Stream Model extends the traditional stream processing model. The transformation function is the core concept and encapsulates user-defined data analytics logic to process data along the stream. These functions are then composed into topologies that enable complex data processing applications. A wrapper is responsible for encapsulating the transformation functions and exposing a thin API layer, enabling the analytics function layer to treat functions as microservices. For stateful functions, these wrappers also provide implicit state

management. The wrapper transparently handles state replication and migration, and access to a function's state is controlled via the exposed API.

The resulting model aims to hide away from developers the heterogeneity of the underlying infrastructure comprising large-scale, heterogeneous pools of resources —from cloud data centres to fine-grained edge nodes. At the hearth of the model lays an orchestrator.It receives the application configuration directives, in terms of high-level objectives such as optimising network latency and decides how to orchestrate the underlying resources, as well as the user-defined functions, by invoking the underlying runtime mechanisms. However, the precise mechanisms for orchestration are left open. The PAPS framework focuses precisely on the orchestration of geo-distributed infrastructures and services through decentralised self-management. Thus, both works are complementary.

Yang et al. [132] tackled the joint placement of independent services onto mobile cloud systems and the dispatching of the workload from various geo-distributed sources. Their system model acknowledges the limitation of each node in terms of how much workload it can serve (computational resource constraint) and how many simultaneous services it can cache (storage constraint). The authors proposed two formulations: a basic service placement problem and a cost-aware service placement problem. The former ignores the cost of deploying and migrating services, whereas the latter is modelled with a multi-objective function comprising these costs. A competitive heuristic to the basic service placement problem is proposed and compared against various benchmark heuristics. An online solution to the cost-aware version of the problem was also proposed and evaluated using taxi and metro mobility traces from a real dataset combined with users access pattern to Youtube videos.

The reported results corroborate the feasibility of the proposed heuristics in solving the basic service placement problem with a low-resolution time. However, the authors did not compare their heuristics against an optimal solution, which prevents us from drawing conclusions about its efficiency. Moreover, the resolution time was reported for up to 80 nodes ($\approx 1.5s$) with a fixed number of 30 services. Given that the complexity of the polynomial-time heuristics is $\mathcal{O}(N^5K^5(N+1)^3)$, one may expect the resolution time to grow considerably for a more significant number of services.

In light of this, the node-level self-management in the PAPS approach could complement the joint service placement and workload dispatching solution by scaling services as the workload fluctuation while a new solution is provided. Conversely, our framework would benefit at the community level from a more robust formulation of the joint service placement

and workload dispatching problem along with the proposed polynomial-time approximation solution. The same rationale holds for the online solution for the cost-aware formulation: the node-level feedback loop could tackle the scaling of containers and thereby mitigate the adverse effects of the workload prediction errors.

Yu et al. [135] proposed a fully polynomial-time approximation for the joint placement and data routing of data-intensive IoT applications in a fog topology. Their solution comprises both single and multiple applications, which are proved to be NP-Hard. The proposed approximation was evaluated with a random topology with 20 nodes, with only 20% of them as fog nodes, while the remaining nodes are exclusively used for networking. Capabilities and requirements for throughput and latency were also randomised for a total of five concurring applications.

The proposed solution was able to improve the QoS of data-intensive applications compared to different heuristics. However, due to the complexity of the problem at hand, experiment results show limitations in terms of scalability. For instance, the designation of hosts for the placement of multiple applications is exponential to the number of applications. Due also to the data routing complexity, the resolution time of the proposed approximation for multiple application placement shows an exponential increase as the approximation parameter is reduced. With an accuracy parameter of 0.3, resolution time was as high as 200 seconds for five applications.

These previous results lead to the following conclusions. Firstly, the scale of nodes with computational capabilities does not correspond to a dense distribution of fog nodes. Secondly, even if the formulation considers multiple applications, each application is monolithic and hosted by a single node. While the formulation is flexible enough to be used with multi-component applications, it would need further assessment for its resolution time, which may be prohibitive for more dynamic workloads. Finally, the proposal focuses on network resources without considering the computational capabilities of each node. This limitation is significant since in the context of densely distributed nodes resources are likely to be constrained.

Zanzi et al. [136] proposed a multi-tenant resource orchestration in MEC systems. Authors introduce a MEC broker responsible for procuring slices of the resources available in the MEC system to various tenants based on their privilege level. At each optimization cycle, the broker decides for placing single-component applications onto the MEC node of choice (gold users) or any feasible node according to the availability of computing resources and the network delay.

We have instantiated our framework with a similar MEC topology. Nonethe-

less, our solution tackles the placement of a dynamic number of instances of various serverless functions onto stateless containers. Hence, while the idea proposed by Zanzi et al. of slicing MEC node resources based on user priority levels is undoubtedly interesting, our framework moves away from user-centric decisions regarding the placement of application components. We approach takes into account the response time as SLA and a varying workload from different sources across the MEC topology.

Nardelli et al. [71] proposed ACD, a general formulation of the deployment and adaptation problem of containerised applications over geo-distributed infrastructures. The proposed model tackles the optimisation of specific deployment objectives with the allocation of containerised application components to geo-distributed VMs and preserves these objectives at runtime through the horizontal and vertical scaling of containers within the VMs. The paper formalises the deployment and adaptation cost and evaluates various deployment heuristics. The paper considers a multi-component application model in which components are independently deployed and scaled, with each instance running in an individual container.

From the operator's perspective, the proposed formulation considers a single application that comprises multiple components. However, the deployment on densely distributed infrastructures must deal with resource contention among concurring applications. Thus, deployment and adaptation decisions from different applications are likely to conflict with each other if taken independently.

From the application provider's perspective, authors rely on a compute-intensive optimisation framework —namely, Integer Linear Programming— for solving the ACD formulation. The authors state that the resulting formulation is NP-Hard. The optimal solution is then compared as a benchmark against three greedy heuristics.

The experimental evaluation was based on simulations involving a five components application. Results are then compared with greedy heuristics which do not take into account the network, which may be considered a threat to the validity of the evaluation. An optimal solution for the more elaborate formulation that minimises both deployment and adaptation costs is found with a resolution time as high as 17 seconds. While this is a valid result for the evaluated application, resolution time could be prohibitive for a higher number of communicating components. Although not tackled by the ACD formulation, the same conclusion can be drawn regarding an extended formulation comprising multiple applications.

The results above corroborate the importance of the decentralised self-management adopted by the PAPS framework: while the ACD formulation

could be employed at the community level, the node-level self-management provides an effective allocation of resources through control-theoretic container scaling.

iFogSim [36] was proposed as a toolkit for modelling and simulating resource management techniques in a Mobile-Edge-Cloud Continuum. It allows one to simulate the placement of different modules of real-time applications to edge devices and then measure latency, network congestion, energy consumption, and cost.

The platform focuses on two types of management decisions: placement of components (application modules) and scheduling. It supports two application models: the sensor-process-actuate and the stream processing model. The platform also comprises classes for the monitoring of the infrastructure and performance prediction.

The Placement and Scheduler classes are the main resource management components. Users may extend their behaviour to implement specific strategies. The Placement class provides out-of-the-box support for *cloud-only* and *edge-ward* strategies. The latter iteratively places application modules, favouring fog nodes in proximity with clients. Upon incompatibility or unavailability of resources, it performs a *leaf-to-root* traversal and places the application module onto the first feasible node. In turn, the default Scheduler behaviour is to uniformly distribute the load among all active application modules.

The EdgeCloudSim [111] is another simulation platform streamlined for edge computing. The platform supports the simulation of multi-tier scenarios where multiple edge servers provide services to mobile user devices in coordination with upper layer cloud solutions.

Similarly to iFogSim, EdgeCloudSim features modular architecture. Besides the functionalities inherited from CloudSim, it provides features such as a mobility model, a load generator, and network modelling for both WLAN and WAN. The platform also provides an extensible orchestrator module in which resource management actions can be implemented. Specifically, EdgeCloudSim supports the creation and termination of VMs, the provisioning of computational resources of edge nodes, and the offloading of tasks to edge or cloud servers based on scheduling policies.

The previous simulation platforms extend the popular cloud simulator framework CloudSim [19]. CloudSim features a centralised architecture. While iFogSim and EdgeCloudSim introduce new features to represent the geo-distribution of servers and clients, they preserve the centralised nature of decision making from CloudSim. In both cases, allocation, placement, and scheduling decisions are performed by a centralised entity (orchestra-

tor). The default implementation consists of straightforward algorithms, whose scalability is demonstrated to be satisfactory. Nonetheless, the resulting performance (e.g. average delay and use of resources) may be significantly lower than provided by optimal, but far more complex solutions. Also importantly, network modelling is limited to the data plane, whereas the control plane is abstracted away —the orchestrator is omniscient of the whole edge system under its control.

While designing the PAPS Simulator, we considered the scalability needed to cope with a large number of nodes in a complex edge system. Not only we tackle the complexity of the decisions involved with the orchestration of geo-distributed nodes and services, but we also consider the challenges of monitoring and analysing the data needed for such decisions, as well as their execution by affected nodes in the system.

These previous requirements lead us to choose PeerSim as the simulation framework for implementing our simulator. As discussed in Section 5.5, PeerSim has been built with scalability as a first-class requirement and facilitates the implementation of the decentralised self-management activities that comprise the PAPS framework.

The PAPS Simulator shares many of the features in EdgeCloudSim, including extensible modules responsible for the generation of dynamic workload, mobility of users, and the modelling of network communication, which in our case includes the control plane (implemented as modular P2P protocols for each self-managing level). In contrast with the orchestration approach used by iFogSim and EdgeCloudSim, the PAPS simulator features the node-level self-management composing our framework with the control-theoretical container scaling as its default implementation —a behaviour that can be disabled or extended.

CHAPTER 8

Conclusions and Future Work

In Chapter 3, we presented a Serverless MEC Architecture to tackle two crucial MEC use cases: latency-sensitive computation offloading and in-transit data processing and analysis. To the best of our knowledge, no other work had employed a serverless architecture in the provisioning of Self-Managed Computing Services by a MEC Platform.

Multiple components may affect the latency perceived by client applications relying on a Serverless MEC Platform for computation offloading. First and foremost, network bottlenecks are the main threat to a client-server architecture targeting latency-sensitive use cases. Hence, new wireless broadband technologies like the *Long Term Evolution* (LTE) and the *5th Generation of Cellular Mobile Communications* (5G) are vital for the success of delivering low latency, high throughput services [24].

The other main threat is platform overhead. The zero preallocation enabled by FaaS comes with the cost of cold start, which may vary from milliseconds (for cold containers) to seconds (for cold virtual machines) [58]. Thus, a Serverless MEC Platform must aim for the balance between resource efficiency, achieved through deprecation of idle resources, and responsiveness, achieved through the retention of infrastructure. In this regard, the client-awareness is paramount for the efficient pre-warming of

containers needed to prevent cold start.

System designers must take into account the particularities exhibited by different functions. Functions that rely on computationally expensive techniques such as image processing and deep-learning require special-purpose hardware for achieving optimal performance. Besides graphics processing units, which nowadays is supported by specific container engines (e.g. Nvidia-Docker [1]), *Tensor Processing Units* have been recently proposed to optimise the training process [2]. According to the OpenFog Reference Architecture [77], hardware accelerators are expected to be deployed at the edge to enable the processing and analysis of large volumes of data.

Last but not least, serverless architectures also provide a different cost model through fine-grained billing. While this billing model is already seen as a positive characteristic of cloud-based serverless platforms, it may become a central feature for driving the adoption of MEC Computing Services among operators. Indeed, it can even complement other utility services (e.g. a mobile data plan) which are often charged in a fine-grained manner (e.g. per minute), hence similar to the serverless billing model.

In Chapter 4, we introduced our vision of a Compute Continuum formed by mobile, edge, and cloud computing. As the central contribution, we presented *A3-E*, a framework for managing the life-cycle of serverless functions deployed to various parts of the Continuum according to application requirements and context.

A3-E overcomes the heterogeneity of the Compute Continuum through mutual client-provider awareness and self-management activities performed by autonomic managers at both sides. Also, it leverages the serverless computing paradigm to allow stateless and lightweight functions to be opportunistically fetched, deployed and exposed by heterogeneous edge and cloud providers, or consumed locally.

The four A3-E activities are distributed across the continuum and conceptually divided into two parts: one is responsible for the autonomic management of function life-cycle, while a client-side proxy is responsible for handling application requests and forwarding them to the provider that can best satisfy the client application requirements.

The feasibility of A3-E has been demonstrated with prototype implementations of its Domain Manager for local-edge (i.e., cloudlets) and Mobile Middleware for Android platforms. Thanks to A3-E, the application was able to autonomously proxy its requests to services that were dynamically deployed to a Compute Continuum. Performed experiments show up

[1] `https://github.com/NVIDIA/nvidia-docker`
[2] `https://cloud.google.com/tpu/`

to a 90% reduction of latency when edge replaced cloud and a 74% decrease of battery consumption when the computation is offloaded from the mobile device to edge/cloud domains. Moreover, by dynamically selecting what constituents to use in the continuum in different contexts, A3-E was able to maximise availability and prevent service interruptions while reducing the overall execution time and battery consumption. Finally, A3-E reduced deployment time up to 70%, compared to a similar approach [120] for the resource management of edge nodes.

Obtained results allow us to conclude that through its *self-management* capabilities, which complement those of serverless computing, A3-E provides a suitable approach for the convergence among mobile, edge, and cloud computing. Nonetheless, from a single MEC operator, A3-E does not tackle the cooperation among MEC nodes from the same operator.

In Chapter 5, we address the previous limitation in A3-E with PAPS, a framework designed to tackle the self-management of densely distributed MEC nodes from a single provider.

PAPS is based on a multi-level, decentralised self-management approach that partitions the larger scale edge topology into delay-aware communities and allocates resources among and within communities. Moreover, the PAPS framework combines optimal community-level allocation and placement with fast node-level container scaling to render the overall self-management process both efficient and effective.

The PAPS framework also comprises a simulation platform. The platform is built on top of PeerSim, a mature P2P simulation framework. The resulting simulation platform allows users to perform experiments with complex topology, workload, and application requirements configurations. The implementation follows a modular architecture. Currently, the PAPS Simulator is integrated with IBM CPLEX —a state-of-the-art solver— and a library implementing the control-theoretic method for SLA-based container provisioning.

8.1 Future Work

Our future work regarding the Serverless MEC Architecture includes the following items.

First, we would like to assess the impact of the function granularity on the software development process.

In a parallel thread, we wish to investigate more complex deployment configurations in which functions from the same application are hosted by heterogeneous MEC nodes according to the node's capabilities and appli-

cation requirements.

We are also interested in tackling the problem of caching data on the edge. While the use of caching is key for mitigating the function initialisation overhead, it is also expensive from the storage perspective. In our current proposal, we considered a least recently used policy. As future work, we wish to evaluate more robust caching policies and techniques.

As a final development, we are interested in the deployment of stateful components. Similarly to the FaaS model, this kind of service would harness the container technology and impose restrictions on the partition size and memory capacity. Differently from FaaS, stateful partitions would privilege vertical scaling and require consistency assurances whenever containers are replicated or migrated.

The roadmap of our future research with the A3-E framework is summarised as follows.

First, we wish to carry out experiments with mobile-edge domains, i.e., with MEC nodes accessed through broadband technology. While the access to cellular infrastructure may be more difficult, we consider that only with a realistic experimental setup we will be able to fully assess the validity of our architectural and management approaches.

Secondly, we would like to deepen our investigation in the context of non-intrusive energy consumption monitoring for leading mobile platforms. More importantly, we wish to equip the mobile domain with the capabilities of a remote domain, i.e., to render it able to host the execution of functions from other devices as part of mobile ad-hoc clouds.

Our research effort to develop a comprehensive framework for tackling the self-management of geographically distributed infrastructures and services from a single provider started more recently. Thus, we envision a series of developments for our work.

First, we wish to perform a robust evaluation of the PAPS approach with various optimisation formulations and control-theoretic techniques. The evaluation of our framework with more complex optimisation formulations is particularly important to highlight the role of the node-level container scaling in preventing SLA violations.

Secondly, we would like to improve our method with the adaptation of the community structure based on the performance of the community-level self-management (e.g. the optimal container placement resolution time). In the same direction, we wish to further investigate the impact of the inter-community allocation in the overall system performance.

Last but not least, we would like to consolidate the implementation of our simulation platform and extend its set of features.

Bibliography

[1] Ahmed H Abase, Mohamed H Khafagy, and Fatma A Omara. Locality sim: Cloud simulator with data locality. *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, 6:17–31, December 2016.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[3] Raian Ali, Nan Jiang, Keith Phalp, Sarah Muir, and John McAlaney. The emerging requirement for digital addiction labels. In Samuel A. Fricker and Kurt Schneider, editors, *Requirements Engineering: Foundation for Software Quality*, pages 198–213, Cham, 2015. Springer International Publishing.

[4] Amazon Web Services. Aws lambda. `https://docs.aws.amazon.com/lambda`, 2019.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[6] K. J. Åström and T. Hägglund. *PID controllers: theory, design, and tuning*, volume 2. Isa Research Triangle Park, NC, 1995.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[8] Timon Back and Vasilios Andrikopoulos. Using a microbenchmark to compare function as a service solutions. In *Service-Oriented and Cloud Computing - 7th IFIP WG 2.14 European Conference, ESOCC 2018, Como, Italy, September 12-14, 2018, Proceedings*, pages 146–160, 2018.

[9] Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen J. Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric M. Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. 2017.

[10] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. A discrete-time feedback controller for containerized cloud applications. In *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 217–228, New York, NY, USA, 2016. ACM.

[11] L. Baresi, S. Guinea, and D. F. Mendonca. A3droid: A framework for developing distributed crowdsensing. In *IEEE International Conf. on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6, March 2016.

[12] L. Baresi, D. F. Mendonça, and M. Garriga. Empowering low-latency applications through a serverless edge computing architecture. In *Proc. of the 6th European Conf. on Service-Oriented and Cloud Computing*, pages 196–210, Cham, 2017. Springer International Publishing.

[13] L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi. A unified model for the mobile-edge-cloud continuum. *ACM Trans. Internet Technol.*, 19(2):29:1–29:21, April 2019.

[14] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. July 2019.

[15] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. 1999.

[16] Michael Till Beck, Martin Werner, Sebastian Feld, and S Schimper. Mobile edge computing: A taxonomy. In *Proc. of the Sixth International Conference on Advances in Future Internet*, pages 48–54, 2014.

[17] Paolo Bellavista and Alessandro Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, pages 16:1–16:10, New York, NY, USA, 2017. ACM.

[18] Flavio Bonomi, Rodolfo A. 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, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012*, pages 13–16, 2012.

[19] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[20] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proc. of the 2010 USENIX Conf. on USENIX Annual Technical Conf.*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[21] 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. ACM, 2010.

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

[23] Tom De Wolf and Tom Holvoet. Emergence versus self-organisation: Different concepts but promising when combined. In Sven A. Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos, and Radhika Nagpal, editors, *Engineering Self-Organising Systems*, pages 1–15, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[24] C. Dehos, J. L. González, A. D. Domenico, D. Kténas, and L. Dussopt. Millimeter-wave access and backhauling: the solution to the exponential data traffic increase in 5g mobile communications systems? *IEEE Communications Magazine*, 52(9):88–95, September 2014.

[25] J. Dolezal, Z. Becvar, and T. Zeman. Performance evaluation of computation offloading from mobile device to the edge of mobile network. In *IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–7, Oct 2016.

[26] Bruno Donassolo, Ilhem Fajjari, Arnaud Legrand, and Panayotis Mertikopoulos. Fog Based Framework for IoT Service Provisioning. In *CCNC 2019 - IEEE Consumer Communications & Networking Conference*, pages 1–6, Las Vegas, United States, January 2019. IEEE.

[27] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. 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, Dec 2013.

[28] Francescomaria Faticanti, Francesco De Pellegrini, Domenico Siracusa, Daniele Santoro, and Silvio Cretti. Cutting throughput on the edge: App-aware placement in fog computing. *CoRR*, abs/1810.04442, 2018.

[29] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[30] H. Flores, P. Hui, P. Nurmi, E. Lagerspetz, S. Tarkoma, J. Manner, V. Kostakos, Y. Li, and X. Su. Evidence-aware mobile computational offloading. *IEEE Transactions on Mobile Computing*, 17(8):1834–1850, Aug 2018.

[31] Ken Fromm. Why the future of software and apps is serverless, 2012. Retrieved from: http://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/.

[32] J. L. Garcia-Dorado. Bandwidth measurements within the cloud: Characterizing regular behaviors and correlating downtimes. *ACM Trans. Internet Technol.*, 17(4):39:1–39:25, 2017.

[33] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[34] Google. Google cloud functions. `https://cloud.google.com/functions/`, 2019.

[35] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Comp. Syst.*, 29(7):1645–1660, 2013.

[36] H. Gupta, D. Vahid, S. Ghosh, and R. Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017. spe.2509.

[37] M. Heck, J. Edinger, D. Schaefer, and C. Becker. Iot applications in fog and edge computing: Where are we and where are we going? In *27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, July 2018.

[38] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, pages 33–39, 2016.

[39] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Openlambda. `https://github.com/open-lambda/open-lambda`, 2019.

[40] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu. Resource provisioning for cloud computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 101–111, Riverton, NJ, USA, 2009. IBM Corp.

[41] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing: A key technology towards 5g. *ETSI White Paper*, 11, 2015.

[42] B. R. Huang, C. H. Lin, and C. H. Lee. Mobile augmented reality based on cloud computing. In *Anti-counterfeiting, Security, and Identification*, pages 1–5, Aug 2012.

[43] IBM. Ibm cloud functions. `https://www.ibm.com/cloud/functions`, 2019.

[44] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe. Evaluation of docker as edge computing platform. In *IEEE Conference on Open Systems (ICOS)*, pages 130–135, Aug 2015.

[45] A. Israel, A. Hoban, A. Tierno Sepulveda, F. Salguero, G. Garcia de Blase, and K. Kashalkar. Open source mano release three – etsi white paper. Technical report, ETSI OSM Consortium, 10 2017.

[46] M. Jia, W. Liang, and Z. Xu. Qos-aware task offloading in distributed cloudlets with virtual network function services. In *Proc. of the 20th ACM International Conf. on Modelling, Analysis and Simulation of Wireless and Mobile Systems*, pages 109–116, New York, NY, USA, 2017. ACM.

[47] J. O Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[48] Florian Klein and Matthias Tichy. Building reliable systems based on self-organizing multi-agent systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems*, SELMAS '06, pages 51–58, New York, NY, USA, 2006. ACM.

[49] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. ThinkAir: Dynamic resource allocation and parallel execution in the Cloud for Mobile Code Offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[50] Chris D. Kounavis, Anna E. Kasimati, and Efpraxia D. Zamani. Enhancing the tourism experience through mobile augmented reality: Challenges and prospects. *International Journal of Engineering Business Management*, 4:10, 2012.

[51] Jay Kreps. Kafka : a distributed messaging system for log processing, 2011.

[52] Karthik Kumar and Yung-Hsiang Lu. Cloud Computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[53] D. Lecompte and F. Gabin. Evolved multimedia broadcast/multicast service (embms) in lte-advanced: overview and rel-11 enhancements. *IEEE Communications Magazine*, 50(11), 2012.

[54] James Lewis and Martin Fowler. Microservices: A definition for this new architectural term, 2019. Retrieved from: http://martinfowler.com/articles/microservices.html.

[55] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief. Delay-Optimal Computation Task Scheduling for Mobile-Edge Computing Systems. *ArXiv e-prints*, April 2016.

[56] J. Liu, T. Zhao, S. Zhou, Y. Cheng, and Z. Niu. Concert: a cloud-based architecture for next-generation cellular systems. *IEEE Wireless Communications*, 21(6):14–22, December 2014.

[57] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[58] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *IEEE International Conference on Cloud Engineering, IC2E 2018, Orlando, FL, USA, April 17-20, 2018*, pages 159–169, 2018.

[59] F. Lobillo, Z. Becvar, M. A. Puente, P. Mach, F. Lo Presti, F. Gambetti, M. Goldhamer, J. Vidal, A. K. Widiawan, and E. Calvanesse. An architecture for mobile computation offloading on cloud-enabled lte small cells. In *IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 1–6, April 2014.

[60] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, Dec 2014.

[61] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *CoRR*, abs/1702.05309, 2017.

[62] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. Live service migration in mobile edge clouds. *IEEE Wireless Communications*, 25(1):140–147, February 2018.

[63] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society.

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

[65] Danilo Filgueira Mendonça, Genaína Nunes Rodrigues, Raian Ali, Vander Alves, and Luciano Baresi. GODA: A goal-oriented requirements engineering framework for runtime dependability analysis. *Information & Software Technology*, 80:245–264, 2016.

[66] Elena Meshkova, Janne Riihijärvi, Marina Petrova, and Petri Mähönen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Computer Networks*, 52:2097–2128, 2008.

[67] Microsoft Azure. Azure functions. `https://azure.microsoft.com/pt-br/services/functions/`, 2019.

[68] Ali Mollahosseini, David Chan, and Mohammad H. Mahoor. Going deeper in facial expression recognition using deep neural networks. In *IEEE Winter Conference on Applications of Computer Vision, WACV 2016, Lake Placid, NY, USA, March 7-10, 2016*, pages 1–10, 2016.

[69] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[70] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive self-adaptation under uncertainty: A probabilistic model checking approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[71] Matteo Nardelli, Valeria Cardellini, and Emiliano Casalicchio. Multi-level elastic deployment of containerized applications in geo-distributed environments. In *6th IEEE International Conference on Future Internet of Things and Cloud, FiCloud 2018, Barcelona, Spain, August 6-8, 2018*, pages 1–8, 2018.

[72] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[73] J. L. D. Neto, S. Yu, D. F. Macedo, J. M. S. Nogueira, R. Langar, and S. Secci. Uloof: A user level online offloading framework for mobile edge computing. *IEEE Transactions on Mobile Computing*, 17(11):2660–2674, Nov 2018.

[74] A. Y. Nikravesh, S. A. Ajila, and Chung-Horng Lung. Towards an Autonomic Auto-scaling Prediction System for Cloud Resource Provisioning. In *Proc. of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 35–45. IEEE Press, 2015.

[75] Nokia Siemens Networks, Intel. Increasing mobile operators' value proposition with edge computing, 2013. Retrieved from: http://www.intel.co.id/content/dam/www/public/us/en/documents/technology-briefs/edge-computing-tech-brief.pdf.

[76] D. L. Olson. *Smart*, pages 34–48. Springer New York, New York, NY, 1996.

[77] OpenFog Consortium. Reference architecture. Technical report, OpenFog Consortium, 09 2017.

[78] G. Orsini, D. Bade, and W. Lamersdorf. Cloudaware: A context-adaptive middleware for mobile edge and cloud computing applications. In *IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 216–221, Sept 2016.

[79] Gabriel Orsini, Dirk Bade, and Winfried Lamersdorf. Cloudaware: Empowering context-aware self-adaptation for mobile applications. *Trans. Emerging Telecommunications Technologies*, 29(4), 2018.

[80] C. Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, May 2015.

[81] Manish Parashar and Salim Hariri. Autonomic computing: An overview. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, pages 257–269, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[82] Ripal Patel, Nidhi Rathod, and Ami Shah. Article: Comparative analysis of face recognition approaches: A survey. *International Journal of Computer Applications*, 57(17):50–69, November 2012. Full text available.

[83] Pawani Porambage, Jude Okwuibe, Madhusanka Liyanage, Mika Ylianttila, and Tarik Taleb. Survey on multi-access edge computing for internet of things realization. *CoRR*, abs/1805.06695, 2018.

[84] Jinmeng Rao, Yanjun Qiao, Fu Ren, Junxing Wang, and Qingyun Du. A mobile outdoor augmented reality method combining deep learning object detection and spatial relationships for geovisualization. In *Sensors*, 2017.

[85] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[86] Jafar Rezaei. Best-Worst Multi-Criteria Decision-Making method. *Omega*, 53:49–57, 2015.

[87] Mike Roberts. Serverless architectures. `https://martinfowler.com/articles/serverless.html`, May 2018.

[88] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio Martín Llorente, Rubén S. Montero, Yaron Wolfsthal, Erik Elmroth, Juan A. Cáceres, Muli Ben-Yehuda, Wolfgang Emmerich, and Fermín Galán. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4, 2009.

[89] Matthias Rohr, Simon Giesecke, Wilhelm Hasselbring, Marcel Hiel, Willem-Jan van den Heuvel, and Hans Weigand. A Classification Scheme for Self-adaptation Research. In *Proceedings of the International Conference on Self-Organization and Autonomous Systems In Computing and Communications (SOAS'2006)*, September 2006.

[90] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Second International Conference on Computer and Network Technology*, pages 222–226, April 2010.

[91] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[92] S. Sarkar, S. Chatterjee, and S. Misra. Assessment of the suitability of fog computing in the context of internet of things. *IEEE Transactions on Cloud Computing*, 6(1):46–59, Jan 2018.

[93] Dimas Satria, Daihee Park, and Minho Jo. Recovery for overloaded mobile edge computing. *Future Generation Computer Systems*, 70:138 – 147, 2017.

[94] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, Aug 2001.

[95] Mahadev Satyanarayanan. A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *GetMobile: Mobile Comp. and Comm.*, 18(4):19–23, January 2015.

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

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

[98] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 815–823, June 2015.

[99] S. Schulte, D. Schuller, P. Hoenisch, U. Lampe, S. Dustdar, and R. Steinmetz. Cost-driven Optimization of Cloud Resource Allocation for Elastic Processes. *International Journal of Cloud Computing*, 1(2):1–14, 2013.

[100] Several authors. Mobile edge computing (mec); terminology v1.1.1. Technical report, European Telecommunications Standards Institute (ETSI), 10 2016.

[101] Several authors. Cisco global cloud index: Forecast and methodology, 2016–2021 white paper. Technical report, Cisco, 11 2018.

[102] Several authors. Mobile edge computing (mec); deployment of mobile edge computing in an nfv environment. Technical report, ETSI GS MEC, 03 2018.

[103] Several authors. Multi-access edge computing (mec); phase 2: Use cases and requirements. Technical report, European Telecommunications Standards Institute (ETSI), 10 2018.

[104] Several authors. Apache openwhisk. `https://openwhisk.apache.org`, 2019.

[105] Several authors. Mobile edge computing (mec); framework and reference architecture. Technical report, ETSI GS MEC, 01 2019.

[106] Several authors. Openfaas. `https://www.openfaas.com/`, 2019.

[107] N. Shalom, Y. Parasol, S. Naeh, and W. Yoram. Nfv and what it means to you: From etsi to mano to yang – cloudify white paper. Technical report, GigaSpaces Research, Cloudify Team, 04 2014.

[108] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[109] A. Sill. Standards at the edge of the cloud. *IEEE Cloud Computing*, 4(2):63–67, March 2017.

[110] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[111] C. Sonmez, A. Ozgovde, and C. Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. In *Second International Conf. on Fog and Mobile Edge Computing (FMEC)*, pages 39–44, May 2017.

[112] Roy Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, Apr 2005.

[113] T. Taleb and A. Ksentini. Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network*, 27(5):12–19, September 2013.

[114] William Tärneberg, Amardeep Mehta, Eddie Wadbro, Johan Tordsson, Johan Eker, Maria Kihl, and Erik Elmroth. Dynamic application placement in the mobile cloud network. *Future Generation Comp. Syst.*, 70:163–177, 2017.

[115] A. u. R. Khan, M. Othman, S. A. Madani, and S. U. Khan. A survey of mobile cloud computing application models. *IEEE Communications Surveys Tutorials*, 16(1):393–413, First 2014.

[116] Luis M. Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Comput. Commun. Rev.*, 44(5):27–32, October 2014.

[117] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, 2017.

[118] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Real-time detection and tracking for augmented reality on mobile phones. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):355–368, May 2010.

[119] Jianyu Wang, Jianli Pan, Flavio Esposito, Prasad Calyam, Zhicheng Yang, and Prasant Mohapatra. Edge cloud offloading algorithms: Issues, methods, and perspectives. *ACM Comput. Surv.*, 52(1):2:1–2:23, February 2019.

[120] Kaiqiang Wang, Minwei Shen, Junguk Cho, Arijit Banerjee, Jacobus Van der Merwe, and Kirk Webb. Mobiscud: A fast moving personal cloud in the mobile network. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, AllThingsCellular '15, pages 19–24, New York, NY, USA, 2015. ACM.

[121] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. Enorm: A framework for edge node resource management. *IEEE Transactions on Services Computing*, pages 1–1, 2018.

[122] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1002–1016, April 2017.

[123] S. Wang, R. Urgaonkar, T. He, M. Zafer, K. Chan, and K. K. Leung. Mobility-induced service migration in mobile micro-clouds. In *IEEE Military Communications Conference*, pages 835–840, Oct 2014.

[124] S. Wang, M. Zafer, and K. K. Leung. Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533, 2017.

[125] Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin S. Chan, and Kin K. Leung. Dynamic service migration in mobile edge-clouds. In *Proceedings of the 14th IFIP Networking Conference, Toulouse, France, 20-22 May, 2015*, pages 1–9, 2015.

[126] Shuo Wang, Xing Zhang, Yan Zhang, Lin Wang, Juwo Yang, and Wenbo Wang. A survey on mobile edge networks: Convergence of computing, caching and communications. *IEEE Access*, 5:6757–6779, 2017.

[127] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. *On Patterns for Decentralized Control in Self-Adaptive Systems*, pages 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[128] J. Xie, B. K. Szymanski, and X. Liu. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *IEEE 11th International Conference on Data Mining Workshops*, pages 344–349, Dec 2011.

[129] Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.*, 45(4):43:1–43:35, August 2013.

[130] Jie Xu, Lixing Chen, and Pan Zhou. Joint service caching and task offloading for mobile edge computing in dense networks. In *IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*, pages 207–215, 2018.

[131] Mengting Yan, Paul C. Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA@Middleware 2016, Trento, Italy, December 12-13, 2016*, pages 5:1–5:4, 2016.

[132] L. Yang, J. Cao, G. Liang, and X. Han. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Transactions on Computers*, 65(5):1440–1452, May 2016.

[133] Ibrar Yaqoob, Ejaz Ahmed, Abdullah Gani, Salimah Mokhtar, Muhammad Imran, and Sghaier Guizani. Mobile ad hoc cloud: A survey. *Wirel. Commun. Mob. Comput.*, 16(16):2572–2589, November 2016.

[134] Paul Leach Yaron Y. Goland, Ting Cai and Ye Gu. Simple service discovery protocol/1.0 operating without an arbiter; internet draft. Technical report, Microsoft Corporation, Shivaun Albright, Hewlett-Packard Company, 10 1999.

[135] Ruozhou Yu, Guoliang Xue, and Xiang Zhang. Application provisioning in FOG computing-enabled internet-of-things: A network perspective. In *IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*, pages 783–791, 2018.

[136] L. Zanzi, F. Giust, and V. Sciancalepore. M2ec: A multi-tenant resource orchestration in multi-access edge computing systems. In *IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, April 2018.

[137] 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, May 2010.

[138] Tianchu Zhao, Sheng Zhou, Xueying Guo, Yun Zhao, and Zhisheng Niu. A cooperative scheduling scheme of local cloud and internet cloud for delay-aware mobile cloud computing. In *IEEE Globecom Workshops, San Diego, CA, USA, December 6-10, 2015*, pages 1–6, 2015.