Universitá degli Studi di Messina

Dipartimento di Ingegneria

Dottorato di Ricerca in Ingegneria Civile Ambientale e della Sicurezza

Curriculum in Scienze e Tecnologie, Materiali, Energia e Sistemi Complessi per il Calcolo Distribuito e le Reti SSD INF/01

Ciclo XXXVI

Applications in the Cloud Edge Continuum: Environment, Challenges and Directions

AUTHOR:

CHRISTIAN SICARI

HEAD OF THE DOCTORAL SCHOOL:

Prof. Gaetano Bosurgi

ADVISOR:

Prof. Massimo Villari

Accademic Year 2022-2023

Abstract

Continuum Computing is a recent term born to refer to the capacity to smoothly integrate Cloud, Fog, and Edge Computing to let applications run wherever they can better exploit the characteristics of the infrastructure to perform better, faster, or more efficiently. Computing at the Continuum is not trivial for many reasons. First, because applications are not thought to be *Continuum native*, but just Cloud or Edge native, and second, because a Continuum infrastructure is composed of heterogeneous and highly distributed nodes, but applications need infrastructure transparency to work properly everywhere in the Continuum. In this regard, the use of containers and orchestrators helps, but those latter are usually heavy and unsuitable for high-constrained Edge devices. Moreover, internet use is huge in highly decoupled infrastructure and even transmitting sensitive data; therefore, keeping Continuum secure is challenging. All those conditions and challenges make the Continuum hard to define, standardize, and implement. In this thesis, we will go through all those aspects that make the realization of Continuum Computing hard, and for all of them, we will propose one or more suitable solutions that will be extensively discussed, designed, and tested. Ultimately, we will even analyze scenarios where Continuum Computing is needed and how our work can help satisfy those needs.

Keywords: Continuum Computing, Cloud Edge Continuum, Serverless Computing, Function as a Service, Serverless Workflows, Osmotic Computing, Urgent Computing.

Contents

List of Figures

List of Tables

Earlier Publications

This thesis is the outcome of the doctoral degree I started three years ago. It is based on selected works (listed below) already published, accepted and under review papers in scientific conferences and journals.

[1] Sicari, C. et al. (2024). Toward the Edge-Cloud Continuum Through the Serverless Workflows. In: Savaglio, C., Fortino, G., Zhou, M., Ma, J. (eds) Device-Edge-Cloud Continuum. Internet of Things. Springer, Cham. https://doi.org/10.1007/978-3-031-42194-5_1

[2] C. Sicari, L. Carnevale, A. Galletta and M. Villari, "OpenWolf: A Serverless Workflow Engine for Native Cloud-Edge Continuum," 2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), Falerna, Italy, 2022, pp. 1-8, doi: 10.1109/DASC/PiCom/CBDCom/Cy55231.2022.9927926.

[3] Sicari, C., Balouk-Thomer, D., Parashar, M., & Villari, M. Event-Driven FaaS Workflows for Enabling IoT Data Processing at the Cloud Edge Continuum The 16th IEEE/ACM International Conference on Utility and Cloud Computing (UCC23).

[4] Gabriele Morabito, Christian Sicari, Armando Ruggeri, Antonio Celesti, Lorenzo Carnevale, Secure-by-design serverless workflows on the Edge–Cloud Continuum through the Osmotic Computing paradigm, Internet of Things, Volume 22, 2023, 100737, ISSN 2542-6605, https://doi.org/10.1016/j.iot.2023.100737.

[5] C. Sicari, A. Catalfamo, A. Galletta and M. Villari, "A Distributed Peer to Peer Identity and Access Management for the Osmotic Computing," 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 2022, pp. 775-781, doi: 10.1109/CCGrid54584.2022.00091.

[6] Christian Sicari, Antonino Galletta, Antonio Celesti, Maria Fazio, Massimo Villari, An Osmotic Computing Enabled Domain Naming System (OCE-DNS) for distributed service relocation between cloud and edge, Computers & Electrical Engineering, Volume 96, Part B, 2021, 107578, ISSN 0045-7906.

[7] A. Al-Dulaimy, C. Sicari, A. V. Papadopoulos, A. Galletta, M. Villari and M. Ashjaei, "TOLERANCER: A Fault Tolerance Approach for Cloud Manufacturing Environments," 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), Stuttgart, Germany, 2022, pp. 1-8, doi: 10.1109/ETFA52439.2022.9921606.

[8] Sicari, Christian & Carnevale, Lorenzo & Galletta, Antonino & Villari, Massimo. (2022). OpenWolf: Serverless Workflow Engine for AI on Continuum. 1st International School on Internet of Things & Edge AI: Computing, Communications and Systems, Sept 8- 12, 2022, Falerna (CS), Calabria, Italy .

[9] C. Sicari et al., "TEMA: Event Driven Serverless Workflows Platform for Natural Disaster Management," 2023 IEEE Symposium on Computers and Communications (ISCC), Gammarth, Tunisia, 2023, pp. 1-6, doi: 10.1109/ISCC58397.2023.10217920.

CHAPTER 1

Introduction

One of the most disruptive innovations in the Information Technology World is represented by the introduction of Cloud Computing. The idea behind Cloud Computing is based on resource virtualization that allows, in a nutshell, hosting multiple virtual infrastructures in one or more real ones, accessing them by the internet, and dynamically changing the available resources. With the promise of saving money, many companies abandoned local data centers to adopt the cloud infrastructure services provided by the major tech big players. Just some years later the advent of the Cloud, the Internet of Things (IoT) rapidly increased in popularity and even its application cases, like smart monitoring, environment monitoring, disaster prediction, and many others. These use cases share a common workflow involving IoT and Cloud: sensors produce data, data are sent to the cloud, the cloud computes the data, sometimes running a machine learning algorithm, then the result is sent back to the stakeholders. In this chain of processes, we can highlight three different downsides:

1. intensive network usage: network bandwidth easily gets congested by the intensive use of it, falling in the impossibility of using it.

2. No real-time computation: sending data to the cloud and waiting for the response has a cost that increases, increasing the size of data we need to transfer and receive, and therefore, the reaction time to some data is really slow, sometimes making the computation useless.

3. Absence of data privacy: mostly for local law constraints, we could need to compute

and store data in a delimited geographic area; by using the cloud, this is never possible.

These limits introduced computation capacity close to the data at the network's edge. This intermediate level has been called Edge Computing and runs lightweight computations, forwarding the heaviest to the cloud infrastructure. For a while, Edge Computing has been considered a good solution to absolve the above-mentioned problems because being located close to the data sources, the network usage is limited, the result is near real-time, and the computation is done in the same domain area of the data source.

From around 2020, the new challenge is letting the cloud and the edge infrastructure smoothly work together, called *Cloud Edge Continuum Computing*, or just *Continuum*. As highlighted in [10], this continuity must be more guaranteed at the application level than at the infrastructure. Continuum native application, in particular, should be:

1. high decoupled: a monolithic application cannot be split into parts by design, so it should entirely run in the cloud or in the edge. High-decoupled applications, usually based on microservices or in Function as a Services (Faas) functions, are naturally split into parts. Each part can be tailored to run in the cloud or edge, depending on the application constraints.

2. As architecture-independent as possible: cloud and edge infrastructures differ in many aspects, like the computation capacity or the network bandwidth. These aspects invalidate the performance of the running application but not the operability. Some other aspects, like the CPU or OS architecture, can make it impossible to run some applications or pieces of them. For this reason, applications should be independent of the infrastructure where they are run.

3. Decoupled from data: continuum applications should be moved among the available infrastructure to guarantee QoS and resource load balancing, but when applications bring data, this process becomes harder. For this reason, the application business logic and the data location should be located in different places to move the business logic without touching the data.

The Cloud Edge Continuum Computing's challenges are the goals of this thesis work; in particular, in this broad scenario, the research questions we focus on are:

RQ1 How to guarantee cooperation in public and private Cloud and Edge infrastructures?

RQ2 How to deploy Continuum native applications?

RQ3 How to guarantee security and privacy in Continuum environments?

RQ4 How to discover and address applications in a Continuum infrastructure?

RQ5 How to orchestrate applications in the Continuum?

RQ6 What are possible use cases of computing at the Continuum?

1.1 Scientific Contributions

In this Section, we will shortly answer the just pointed research questions. Then, for each question, we will forward to the right chapters to extensive read about our solutions and related research.

RQ1: *How to Guarantee Cooperation in Public and Private Cloud and Edge Infrastructures?*

Contribution: defining a guideline architecture to design and build Continuum native applications

We learned that the FaaS is a good approach to deploy applications on the cloud and on the edge, and in particular, the Faas workflows are a good option to make these applications *continuum native*. On the other hand, we ignored that vendor-locking producers mostly provide cloud infrastructures and serverless services, making it harder to integrate with other providers or open-source solutions. Considering these issues, we tried to provide a common architecture for serverless cloud-edge workflow suitable for private and open-source solutions, aiming to make integrating these different contexts easier. We extensively answered this question in Chapter 3.

RQ2: *How to deploy Continuum native applications?*

Contribution: Composition of FaaS-based application orchestrated in heterogenous infrastructures

We can affirm that Docker convinced the industrial and scientific community that the best solution to deploy applications *everywhere* passes through the Linux containers. Linux containers, nowadays, are no longer deployed on bare metal servers, but orchestrators like Kubernetes, Mesos, and Nomad are used to deploy them and control their life-cycle over a clusterized environment composed of cloud as well as edge infrastructures. Due to the complexity these orchestrators reach nowadays, Serverless and, specifically, FaaS have been put on top of them to do these jobs. We have seen in containers, orchestrators, and serverless the opportunity to build multi-architecture applications able to be spread, synchronized, and relocated in the cloud and the edge, guaranteeing, in the end, the *continuum* of the

service. Even with those tools, we still have to find a way to design complex and Continuum Native applications by binding functions. In Chapter 4, we proposed two ways to do that. The first one, called OpenWolf is the first open-source scientific workflow serverless engine based on Kubernetes, OpenFaas, and other open-source solutions. OpenWolf allows defining somewhat *industrial* workflows, which means the steps inside are well-defined and known, and then an initial raw object is processed following the workflow. On Continuum, workflows can be prone to dynamicity, which means who produces, who consumes, and how to consume data continuously changes. OpenWolf cannot shape these characteristics. Because of that, we proposed a second solution based on the RPulsar engine, extending it to work on Continuum using FaaS. This engine, unlike OpenWolf, lets bind functions by the dynamic match of profiles between producers and consumers. Each binding shapes a workflow in the Continuum.

RQ3: *How to guarantee security and privacy in Continuum environments?*

Contribution: applying the Osmotic Computing principles to the deployment systems we adopted in RQ2.

There are many security aspects to consider when dealing with applications on the cloud and edge. From an application perspective, Authentication, Authorization, and Accountability (AAA) are the fundamentals needed to guarantee the minimum requirements of a basic trustable service. In distributed and connected environments, other aspects must be considered, like network security or the transmission of sensitive data. We tried to cover all those aspects, proposing two works described in Chapter 5. In particular, firstly, we tried to secure the infrastructure and the code execution. We did that by improving the OpenWolf project using the Software Defined Membrane (SDMem), to keep secure connections and access to private data. Then, we addressed the Authorization and Authentication issues in unstable Continuum environments, where policies can continuously get updates and then must be propagated. We did that by proposing an Osmotic Identity Manager, eventually consistent among a Peer IDM Network, and we integrated it into the previous Osmotic Infrastructure.

RQ4: *How to discover and address applications in a Continuum infrastructure?*

Contribution: Proposing a geohash-based DNS to locate and discover services

In some dynamic environments, such as smart cities, services provided to the citizens are dynamic, related to a specific area or target, then prone to change or *move*; for that reason, locating and reaching them can be hard. In this scenario, we proposed a geographical-based DNS, called OCE-DNS. This distributed infrastructure based on the principles of the Osmotic Computing that will be discussed later is described in 6, and it allows exploring and reaching

out to services, applying DNS standard queries using three-dimensional geo hashes as FQDN.

RQ5: *How to orchestrate applications in the Continuum?*

Contribution: using and customizing container-based orchestrators

Container orchestrators are the standard de facto to deploy and manage cloud applications, but this is not true on the edge. In highly constrained devices, orchestrators are prone to fail or overload the devices, making it impossible to run further applications on them. For these special environments, where the edge infrastructure is abroad but constrained, we developed Tolerancer. Tolerancer is a peer-to-peer light orchestrator composed of two units called Monitoring and Middleware Unit; the latter makes possible the communication and the sharing of information between devices, and the first one analyzes and reacts to failures or overloads in the system. This project is argued better in 7.

RQ6: *What are possible use cases of computing at the Continuum?*

Contribution: providing different use cases where computing at the Continuum increases the quality of the service

In recent years, computing at the Continuum has become a hot topic due to the several use cases involving cooperation between the Cloud and the Edge. In Chapter 8, we discussed those use cases and presented two works. In the first one, we used OpenWolf to deploy and manage a deep learning pipeline to train and analyze images from smart cameras installed on a city street in real-time. The second work focused on Urgent Computing use cases like river flooding, forest wildfire, etc. Those themes are the main focus of the European Project TEMA, which aims at providing efficient systems to prevent and control the territory. We contributed to this project, designing a highly decoupled heterogeneous architecture to efficiently analyze data from IoT sensors using serverless workflows.

This thesis's topics, challenges, and solutions are summarised in Figure 1.1.

1.2 Structure of the Thesis

The Thesis structure tries to link each research question to a chapter to improve readability. In Chapter 2, we provide the background knowledge needed to understand the further work, then the core of this thesis work is discussed between Chapters 3 and 8, in particular:

In Chapter 3, We proposed a guideline architecture to design build and deploy Continuum Native applications.

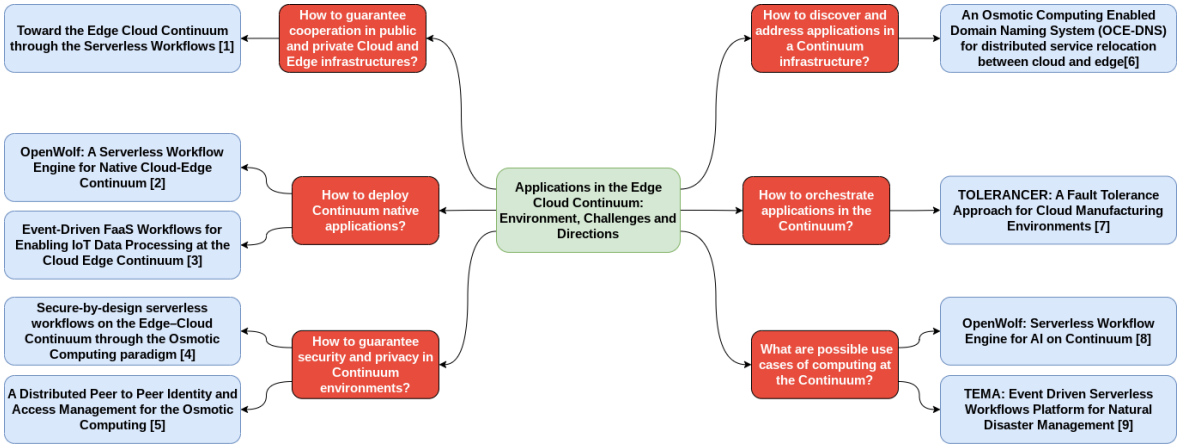


Figure 1.1: Research questions and presented works mind map

In Chapter 4, we discuss OpenWolf and Enhanced RPulsar to deploy applications on the Continuum.

In Chapter 5, we discuss the security threats regarding deploying applications on the Continuum and propose Osmotic Computing solutions to mitigate them.

In Chapter 6 we propose an Osmotic computing-based solution to locate and relocate applications on Continuum.

In Chapter 7 we discuss Tolerancer, a micro orchestrator for applications running on constrained infrastructures.

In Chapter 8, we wrap up all the works, discussing two real scenarios where our solutions have been used.

Each core Chapter has an introduction to what we want to address there and includes one or more works. Each work is organized using a common structure as follows: 1. introduction; 2. review of the state of the art; 3. discussion about the background and specific technologies; 4. design and implementation of the system; 5. performance evaluation; 6. conclusion summarising specific contribution.

Finally, Chapter 9 concludes this thesis work and lights the future research directions.

CHAPTER 2

Background

This thesis aims to provide solutions to enable the Cloud Edge Continuum, that is, the ability to guarantee the continuity of the computation among different computation layers. Before doing that, we must discuss the building blocks that drove us to the Continuum's challenges. For this reason, we shortly introduce some important concepts, such as Cloud, Edge and Fog Computing, Serverless Computing, and Osmotic Computing. All those terms will be extensively used, and to better understand our work, it is important to understand how they work, what advantages they provide, and their disadvantages and issues.

2.1 Cloud, Edge and Fog Computing

Cloud Computing is the evolution of traditional data centers; it allows to virtualize hardware and software resources, such as servers, storage areas, and networks that can be provisioned on-demand with different quality of service and cost in remote data centers. Historically, Cloud Computing is characterized by five key points, that are: • on-demand self-service; • broad network access; • resource pooling; • fast elasticity; • measured service; The service models that a cloud provider are many, and they continuously increase, but basically, the officially recognized ones are:

- Software as a Service (SaaS): which provides out-of-the-box applications such as remote storage;

- Platform as a Service (PaaS): which provide platforms that let host specific kind of applications, like, for example, Heroku to host code;
- Infrastructure as a Service (IaaS): which provides resources used to build virtual machines, VPS, storage areas, and so on;

Cloud Computing is powerful because it makes it possible to provide high-performance resources at any time, paying just the cost of the real, but this model does not fit use cases where it is still important to bring the computation as close as possible to the data, reducing network latency and improving the response time. Edge Computing is thought to absolve these issues. In edge computing, the storage and computation are performed on devices located at the network's edge rather than on a centralized server or in the cloud. Edge computing can be used with cloud computing, with the cloud being used for more resource-intensive tasks and edge devices handling more localized, real-time processing.

Fog Computing has been recently located in the middle between Cloud and Edge. Fog computing typically acts as a broker that deploys computing resources at the network edge and connects them to a central cloud-based infrastructure. In some cases, the Fog layer can even act as a middle computation layer, providing near real-time computation to the edge infrastructure without involving the cloud. Typically, the Cloud, Fog, and Edge are represented in figure 2.1

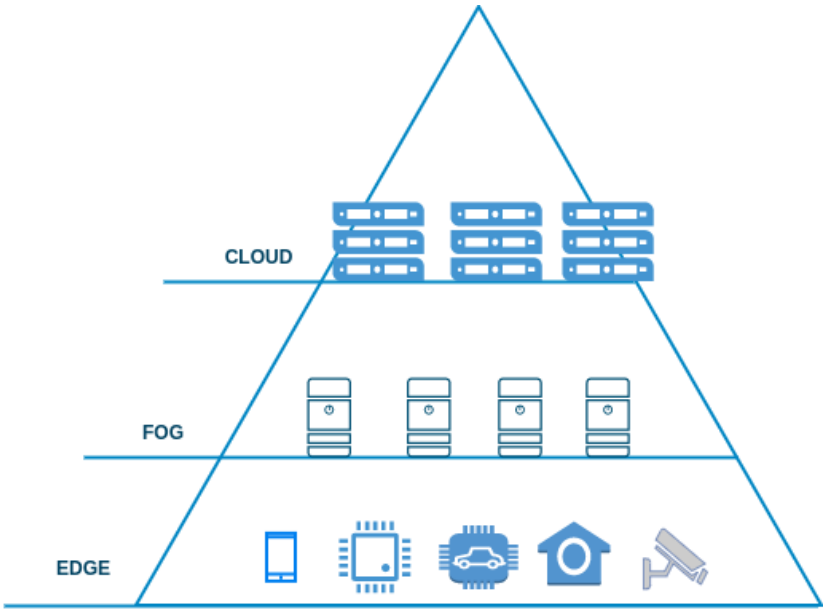


Figure 2.1: Cloud, Fog and Edge pyramid

2.2 Cloud Edge Continuum

As we understand from the previous section, Cloud, Fog, and Edge are different computation and storage layers organized in a hierarchy mainly based on their location. These three layers are not supposed to be traversed in a specific order, and they are not supposed to be used the time together; some applications could require using just one or two of them, and the choice may change over time, depending on many different factors like the required quality of the service, the emergency of the obtaining a result, or the traffic in the network. The Cloud Edge Continuum challenge is to spread, locate, relocate, and synchronize applications to where they are needed. Continuum, in some way, wants to split the application parts from which they are executed to have a unique big computation infrastructure that is used dependently on the current needs. Unfortunately, there are no unique solutions or standards to create a Continuum environment, but there are some paradigms, like Osmotic Computing, and some architectural models, such as the microservices and the Serverless, that can be exploited, as we will see, to create it.

2.3 Microservices and Orchestration

The Microservice architecture is a software design model that structures an application as a collection of loosely coupled, independently deployable services. Each microservice is in charge of a specific task while interacting with other services using light communication protocols. Microservices can be deployed, scaled, and updated independently, making building, testing, and maintaining complex systems easier, especially using some containerization tools, like the Docker containers, that can inbox an entire service in a unique and independent unit.

However, managing microservices can become challenging as the number of microservices grows. At this point, orchestration is used. Microservice deployment, scaling, and interaction coordination are all handled automatically through orchestration. Utilizing a system that can automatically plan and manage the deployment of microservices across a cluster of servers, such as Kubernetes or Docker Swarm, is often required.

Managing microservices' configuration, keeping an eye on their well-being, and taking care of service discovery and routing are all part of the orchestration. Orchestration can assist in ensuring that microservices are consistently available and operating properly, even if the application changes over time, by automating these processes.

2.4 Serverless Computing

Function as a Service (FaaS) and Serverless Computing, are cloud computing architectures in which the provider administers the infrastructure and automatically allots and adjusts the resources required to operate and manage specific functions or applications. Thanks to serverless computing, developers may concentrate on building code without worrying about the supporting infrastructure.

Code is run in a serverless architecture responding to events like HTTP requests, database updates, or queued messages. Each function runs in its own container, which the cloud service provider immediately creates when the function is invoked. The function runs only for the length of the event, and the cloud provider deallocates the container when the function completes. Serverless computing can make the development process simpler, which is another advantage. Developers can concentrate on writing code and creating applications because they do not have to worry about administering servers or infrastructure. This can assist businesses in reducing time to market and raising the general caliber of their apps.

However, there are also drawbacks to serverless computing, including vendor lock-in, a greater need for expertise in maintaining distributed systems, and possibly higher operating expenses for particular workloads.

Serverless computing is considered a natural evolution of microservice architecture because it increases the concept of "decoupled application", but unlike microservices architecture, the presence of an orchestrator is required to let it properly work.

2.5 Osmotic Computing

Osmotic computing is a new computing paradigm that seeks to enable seamless, dynamic resource sharing across multiple devices and clouds. In osmotic computing, devices and clouds are connected through channels that enable the exchange of computational resources, such as processing power, memory, and storage.

Osmotic Computing aims to standardize what is currently happening with orchestrations and Faas but still improve it, letting application be *Continuum native*.

The concept of osmotic computing is inspired by the natural phenomenon of osmosis, where fluids move across a permeable membrane to achieve equilibrium. In osmotic computing, computational resources are similarly exchanged between devices and clouds to achieve a balance of resources across the network.

One of the key benefits of osmotic computing is that it can help address the challenges of traditional cloud computing, such as data latency, network congestion, and security concerns. By enabling dynamic resource sharing across devices and clouds, osmotic computing can improve performance, reduce costs, and enhance data privacy and security.

Osmotic computing is still an emerging area of research, and many challenges must be addressed before it can be widely adopted. These challenges include developing new algorithms and protocols for resource allocation and management, addressing security and privacy concerns, and ensuring compatibility across different devices and clouds.

Despite these challenges, osmotic computing has the potential to revolutionize the way we think about the Cloud Edge Continuum and enable new applications and use cases that were previously impossible. With its focus on seamless, dynamic resource sharing, osmotic computing could pave the way for a new era of distributed computing that is more efficient, resilient, and adaptable than ever before.[11]. As the name itself suggests, this paradigm is inspired by the chemical phenomenon of osmosis, which consists of the diffusion of the particles of a solvent from the region with its highest chemical potential to that with the lowest chemical potential when the diffusion of the solute is prevented by means of a semi-permeable membrane.

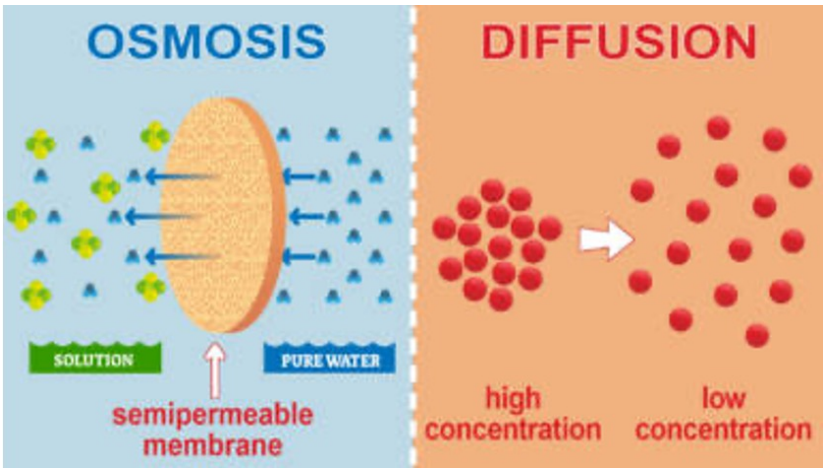


Figure 2.2: Osmosis phenomenon

Similarly, microservices should move within an Osmotic Membrane in Osmotic Computing to find a configuration that satisfies given requirements (i.e., quality of service). Osmotic Computing defines a lot of terms and concepts, but we mainly focus on (i) MicroElements (MELs) and (ii) Membrane. The first ones are software or data abstractions and are classified according to their nature [12] of MicroService (MS) or Microdata (MD), as shown in Figure 2.3. They are furthermore divided into MicroOperationalServices (MOS) and MicroUserSer-

vices (MUS) and between MicroOperationalData (MOD) and MicroUserData (MUD). More specifically: (i) MOS can be associated with operative systems; (ii) MUS can be associated with user applications; (iii) MOD stores configurations for MSs; and (iv) MUD stores user data.

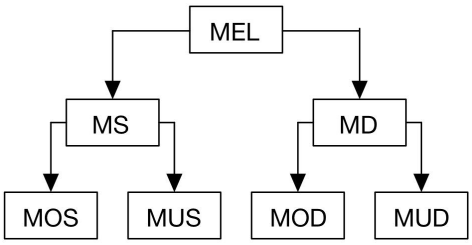


Figure 2.3: MELs classification.

While MODs are usually sharable between different applications, MUDs are designed to be accessed by the owner (i.e., a user, an MS, or an application) to ensure privacy. Osmotic Computing defines a concept of intra-membrane that isolates the MS/User space and MUD from the outside, satisfying the objectives O1 and O2.

The Osmotic SDMem is a virtual environment based on the underlying infrastructure (Cloud or Edge resources). Inside this environment, MELs are isolated from the rest of the world. Indeed, they can migrate through the osmotic nodes without interacting with external infrastructure. The Osmotic Membrane acts as a filter to limit how MELs can be migrated and under which constraints [13].

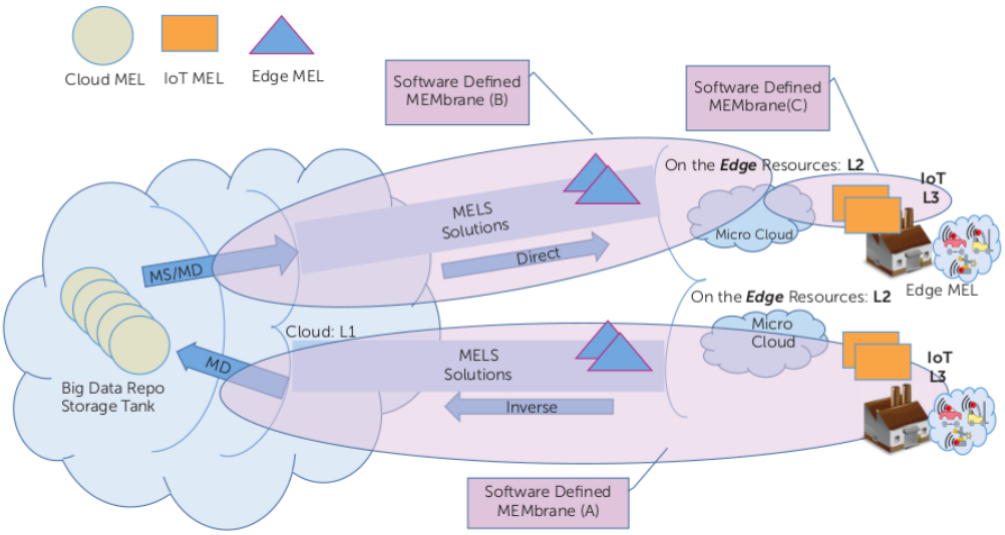


Figure 2.4: SDMem in Osmotic Computing

SDMem allows each organization to enable the grouping and filtering of MELs[14] based on their properties and purposes when organized like a federated ecosystem. SDMem allows

MELs to migrate according to constraints identified in the membrane, guaranteeing isolation of one system from another [15]. An SDMem is instead a security component that Osmotic Computing introduces to guarantee privacy and confidentiality for authorized interactions. Firstly, a MEL can interact only with the MELs in the same SDMem, satisfying the Objective O3. Moreover, the SDMem guarantees a network privatization that isolates the connections from the rest of the cluster, satisfying the Objective O4. Ultimately, any point-to-point message is confidential by design, which means that another MEL in the same SDMem cannot access it, satisfying the Objective O5. Thus, SDmem allows isolating nodes in a federated environment; in that way, MELs can only move through other nodes that are part of the same SDMem. At the same time, it allows the isolation of different membranes (i.e., creating computation context) and, therefore, the MELs of which it is composed to prevent unexpected interaction.

CHAPTER 3

Guaranteeing Cooperation in Public and Private Cloud and Edge Infrastructures

FaaS is emerging as the prominent solution for making simpler, faster, and architecture-independent software deployment on Cloud and Edge tiers. This new trend has partly been used in Cloud-Edge Continuum applications that need to run functions over different nodes to respect some defined QoS parameters, like data closeness or high-performance computing. Unfortunately, FaaS has some lacks that hardly allow adding the 'Continuum' aspect to an Edge-Cloud environment due to the absence of a transparent architecture environment, a tier-aware scheduler, and mostly a capacity to combine in a complex relationship different functions in a Continuum native serverless workflow. Until now, Cloud and Edge providers can offer solutions to deploy and run functions, and some can even build similar workflows. Unfortunately, those providers did not share a standard or a guideline architecture; therefore, integrating them to build hybrid workflows is somewhat impossible. We aim to provide a reference architecture for deploying, composing, and making environment-aware serverless workflows composed by FaaS functions spread over the Continuum.

3.1 Introduction

Deploying software at the Continuum is considered challenging for many reasons, such as architecture dependency, host federation, and global resource balancing, [16], [17]. However, the serverless paradigm was recently born to make these problems surmountable. Faas

engines are typically based on an orchestrator (i.e., Kubernetes), which is able to manage multi-architecture containers and load balance and federate resources [18]. Serverless and FaaS paradigms are widely used in cloud-only applications, but thanks to their flexibility, some recent works are emerging with the purpose of deploying functions into the edge of the network for lightweight problems [19], [20], [21]. For example, FaaS can be used for isolated and low-decoupled tasks, but it is not ideal for complex and coupled applications due to the impossibility of easily composing and integrating functions [22]. These drawbacks generate issues for continuum environments where, typically, applications are coupled in data-driven workflows with many tasks connected among different computing tiers [23, 24, 17].

In this chapter, we propose i) new research guidelines for serverless orchestration in the cloud-edge continuum paradigm and ii) a reference blueprint for the standard creation of a FaaS-based workflow orchestration. Specifically, we determine principles, definitions, a reference architectural model, and data structures that are useful for defining and orchestrating serverless workflows. This baseline architecture will be used to design OpenWolf, which is described in Section 4.1, and it will be used in Section 8.1 to deploy a deep learning workflow for image classification in a Smart City scenario, considering five steps: (i) collection, (ii) transformation, (iii) training, (iv) inference, and (v) plotting.

3.2 Background

The Continuum Computing aims to make a collaboration between the cloud and edge tiers in order to distribute near real-time processing on the edge and massive processing on the cloud [25]. Continuum faces several challenges related to different topics (i.e., security, scheduling) such that actual solutions [10] need to be re-engineered to become suitable for the Continuum Computing.

Recently, serverless computing has emerged as a solution for distributing small functions using containers intending to react to external triggers (i.e., cronjobs, HTTP calls, message queue systems) [26]. This new paradigm was well received by the scientific community, which tries to exploit it for orchestrating functions over the Continuum Computing [27] by using different orchestrators, such as Kubernetes, [28], Nomad [29], [30], and more [31]. Moreover, FaaS is used in the Continuum Computing to make development, deployment, and automatic balancing easier thanks to the underlined orchestrators [32, 33, 34].

The combined use of cloud-edge continuum and serverless pointed out the problem of composing functions, which means the capacity of concatenating functions for creating more

complex applications. Authors [35] proposed three principles of serverless as (i) black-box functions, (ii) substitution, and (iii) double billing, which attempt to explain that composing FaaS application could be considered an anti-pattern. However, we do not agree with that statement.

The term workflow was used as a generic term for describing a well-defined organization of tasks connected in order to transform one or more inputs to a given output. In scientific literature, this term muted to *Scientific Workflows*, which is described [36] as a way to deal with data and pipelined computation steps in different application fields (i.e., bioinformatics, cheminformatics, ecoinformatics, geoinformatics, physics), without mastering a computer science background. For example, Kepler is a workflow grid-based, later extended [37] to support distributed computing on grid computing. Almost in parallel, the Pegasus system [38] was proposed to abstract the workflow as an ensemble of independent tasks. Such technology continued to have a progressive evolution, keeping track of newer ones, such as grid [39], [38], cloud [40], containers [41], and [42]. Going towards the last five years, workflows gained new popularity because of the increasing use of cloud computing and Serverless. Indeed, the latter was widely adopted for designing and implementing workflows [43]. Perez et al. [44] designed a framework for executing Linux-based containers in a FaaS platform (i.e., AWS Lambda). Jiang et al. [43] integrated the scientific workflow into the main FaaS providers in order to exploit the serverless paradigm and make easier the implementation for end users (i.e., scientists). Skyport [45] was instead a brilliant idea for creating black-box-based workflows, by means of an engine able to compose workflows as soft virtualized software (i.e., Docker containers). Recently, the workflow has become more sophisticated and accurate. It is not a programming pattern or a software architecture design, but a computational on-premise engine that defines, stores, and deploys a composition of black-box functions [46]. Hyperstream [47] is a domain-specific tool to deploy Machine Learning (ML) algorithms that are automatically fired by some incoming streaming data. One step ahead in this direction was moved in [48], where the authors proposed a Workflow Engine Server (WES), which is a back-end engine used to store functions and workflows and run them when triggered by an event. Such an engine introduces workflow modularity and a validation schema, but it lacks integration with external systems and expandability with other functions. One of the most autonomous engines has been instead presented by Lopez et al. [49] with Triggerflow, a trigger-based orchestration of serverless workflows. It lacks a user-friendly workflow editor, a data schema for the functions, and an event global registry. However, Triggerflow has clear strengths, such as a mechanism to fire triggering-based

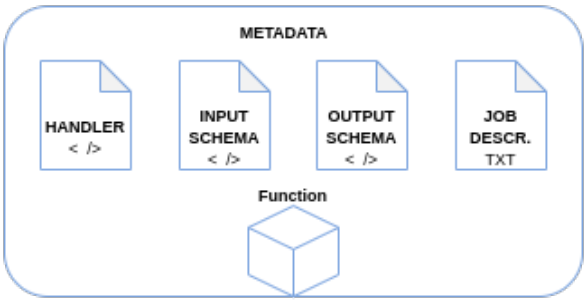


Figure 3.1: State structure

workflow, an asynchronous communication channel, and a serverless model. A different approach for workflows was, instead, presented in [50], where authors propose R-Pulsar, a cloud-edge engine able to trigger functions according to an interesting matching algorithm based on a decoupled Associative Message (AR) selection already presented in [51]. This helps in matching producers and consumers, as well as taking actions, such as running a function and starting a data production [52]. The above-mentioned approaches prove good flexibility mostly when related to ML [53], but the utilization of serverless is still not totally well exploited.

3.3 Workflow Engine Characteristics and Principles

In this Section, we put the stakes of the proposed workflow engine architecture, and we define the dictionary of terms that are used in the remainder of this Chapter, i.e., i) state, ii) event, iii) workflow, iv) manifest

3.3.1 State

The main component of the architecture is the *state*. It mainly encapsulates a function and all the information related to it. It is stateless, which means the running state is unaware of other states interacting with it, and therefore, the state behavior can not change based on previous executions. As shown in Figure 3.1, the state is composed of i) *metadata* and (ii) a *function*. The latter code includes the state's business logic and is encapsulated inside a container.

The metadata includes four pieces of information: state description, handler instructions, input, and output schema. Specifically, they are described as follows:

State Description contains the state identifier, name, service description, and service class. They are used to classify the service quickly.

Bootstrap Instructions are run for instantiating a state inside the Workflow Engine. These could contain the code to build an image, set the environment variables, or run a docker container.

Handler instructions are run every time a state is triggered. These validate the input schema, run a function using the passed and parsed parameters, wait for the function result, and finally parse results with a format compliant with the output schema.

Input/Output Schemas contains the schema of the acceptable input and the schema of the provided output. They are essential for creating compatible state chains.

Often, Workflows also contain the *connectors*, a special state that simply maps a state's output to the next states' input, according to their input/output scheme. It is created on-premise during the workflow design, and it does not require an input and output predefined schema since they change according to the workflow where they are located.

3.3.2 Event

An *event* is the only entity that can be processed in a workflow; it is originally sent from outside and then processed inside the workflow. All changes applied to an event are separately stored in a data lake, while the last version of the event is propagated through the workflow states. An event is composed of both immutable and mutable data. The immutable data includes:

Event ID identifies the event uniquely and is managed directly by the workflow engine.

Workflow ID refers to the workflow that is processing/has processed the event.

The mutable data are generally updated by the workflow engine and by the states that process the event. This includes:

Status is a value in the following domain: ⟨ Started, Processing, Error, Processed ⟩.

Data is the last state's output.

Timestamp represents the date and time the last transformation has been completed.

3.3.3 Workflow

The workflow diagram in Figure 3.2 represents how states interact. The workflow starts when the first node is triggered by an external event, i.e., action 1 in Figure 3.2, carrying on a

data payload. Any event is directly connected to a *State* (action 2) and therefore to a *Connector* (action 3). Connectors act as conciliators for filtering events with a specific state.

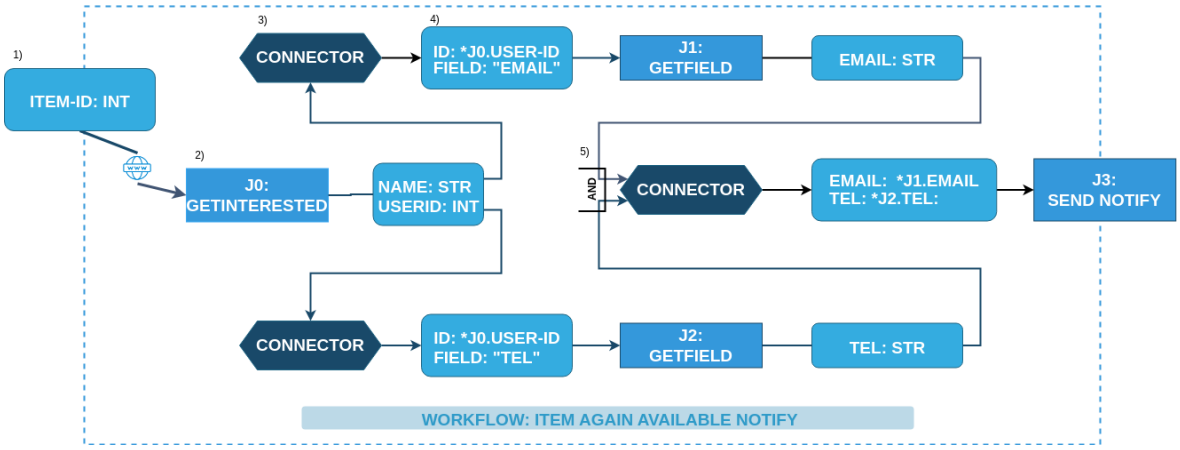


Figure 3.2: Workflow example

The first event is unique and mapped one-to-one to a single workflow execution. This avoids overlaps with other events that follow the same workflow. Naturally, when an event passes through the *States*, it modifies its data according to the output of the previous state.

Any link is allowed within the workflow, such as many-to-many, many-to-one, one-to-many. However, they must start and finish with only one *state*. The triggering condition must be explained when a many-to-one relationship (action 5 in Figure 3.2) is defined. In this regard, the condition may follow the boolean algebra, i.e., using AND for combining two or more events that must be received before firing the next one or using OR for combining two or more events according to the fact the only one of them is enough for firing the next state.

The workflow diagram shown in Figure 3.2 is an example of an e-commerce scenario, where customers are notified by email and a short message system as soon as a product they are interested in is again available. Furthermore, the workflow is triggered by a web notification that says a given product is available again. The workflow fetches the users interested in this item using the *State* J0, J1, and J2, then fetches the users' email and telephone numbers. Finally, the *State* J3 is used to notify the users. In this scenario, three connectors are used. Two connectors make the J0's output compatible with the J1 and J2's input. The last one maps the J1 and J2's output instead of the J3's input.

3.3.4 Workflow Manifest

To describe a workflow within a schema, we propose a manifest based on YAML format. The manifest translates in processes what was designed, i.e., in Figure 3.2.

```
name: <workflow-name>
callbackUrl: <uri-where-to-send-result>
states:
  <state-id>:
    function:
      ref: <ref-to-function-id>
      config:
        key: value
    start: true
handlers:
  <handler-id>:
    endpoint: <endpoint-to-function>
    config:
      key: value
workflow:
  <state-id>:
    activation: <Boolean Equation>
    inputFilter: <jq command>
    outputFilter: <jq command>
```

Listing 3.1: Workflow Manifest example

As shown in the listing 3.1, the manifest has i) a name, ii) a callback URL where sending the result, and three more sections, such as iii) States, iv) Handlers, and v) Workflow.

States lists and describes all the statuses of the workflow. For each state, we define a name, handler, and a global key-value configuration for the handler.

Handlers describes all the handlers called within the states. This attribute determines how to call the handler and the basic configurations that may be overwritten in the States' parts. The separation of States and functions sections allows multiple times the same handler in different states.

Workflow describes how the states interact. We determine which previous states have triggered each state and how to transform inputs and outputs. This part acts as a connector.

3.4 Architecture

Figure 3.3 shows the reference architecture for managing a serverless workflow. It is a four-layered architecture composed of i) infrastructure, ii) federation, iii) serverless, and iv) service layers. All layers are described as follows.

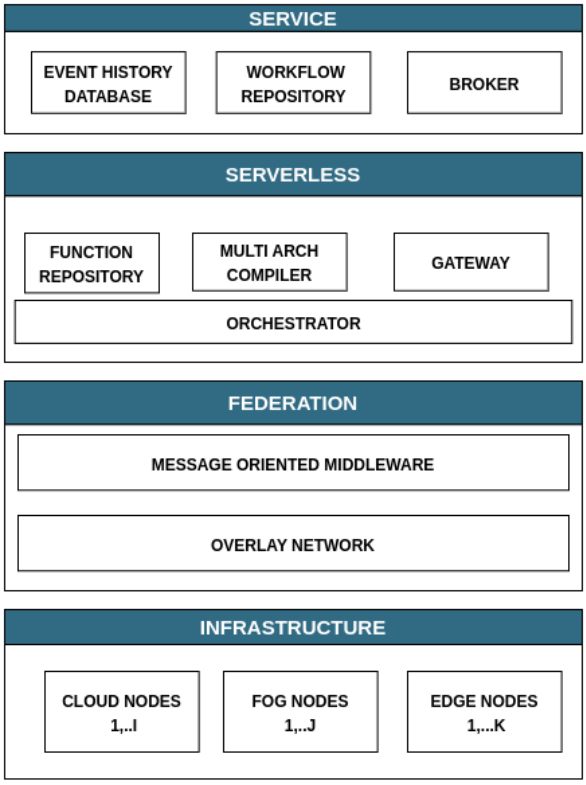


Figure 3.3: Workflow Engine architecture

The *infrastructure layer* contains the bare-metal nodes in the Continuum Computing environment. Nodes may have different geographical locations, architectural characteristics, and distribution. The *federation layer* creates communication interoperability among the nodes of the infrastructure layer. It comprises an overlay network that connects nodes with a Message Oriented Middleware (MOM), intending to exchange data over the overlay itself. The *serverless layer* provides FaaS features to the underlined layer, i.e, the service layer. It uses a container orchestrator for deploying functions among the federation. It includes a function repository for storing the functions in the system, a compiler to build the same function in all the architecture available and compatible, and a gateway used to trigger the functions. The *service layer* is, instead, the top layer of the architecture. It adds the capability of composition to the serverless layer. The service layer is composed of an Event History Database (EHD), a Workflow repository, and a single agent. The EHD stores a permanent history of events transformation within the engine. Indeed, an event changes its mutable content when it is the input of a state. However, if a workflow is composed of n-states, the initial event will have n changes. Thus, the EHD stores all the n changes, along with the initial content. Furthermore, we had to consider a Status History array field in the event data structure, as shown in Figure 3.4. This approach allows to: i) keep track of the event history, ii) keep track

of the event transformation, iii) log every change, and iv) recover any workflow state. The *workflow repository* stores the manifests files that contain the workflow descriptions according to the structure defined in Section 3.3.4. The *Broker* coordinates the service layer and, more in general, the overall infrastructure. It basically is in charge of receiving the external events and intercepting the execution of a function inside a triggered state, recognizing that it uses the proper workflow manifest in the workflow repository, and then updating the EHD for saving the actual data coming from the events or the states.

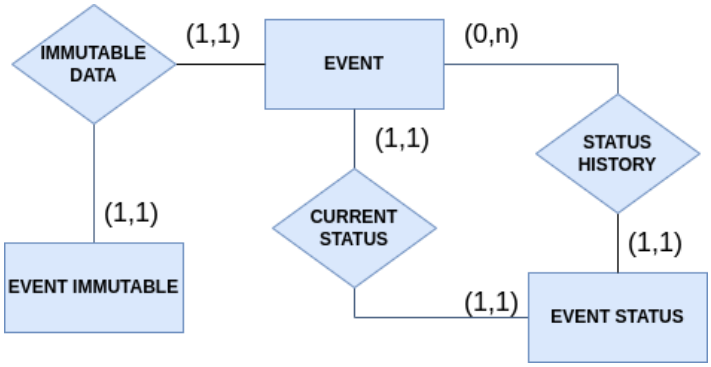


Figure 3.4: Event data model

3.5 Conclusion

In the era of serverless and microservice architecture, workflows are slowly going to gain popularity as a tool to mix serverless services and deploy them in order to compose complex functions in modern engine infrastructure based on the Cloud.

Historically, workflows are recognized as a computation chain where the processes involved depend on the specific field where they are acted. In the last two years, this term has started to appear in different fields, like microservices, FaaS, and cloud-edge continuum. In some way, the scientific community shares the idea that workflows enable the cooperation between functions, services, and in general network hosts.

This trend is fully reasonable since we managed to deploy functions and services everywhere, just to think of the new concept of the ”Internet of Everything“. However, we did not manage to link these capabilities together. To try to reach this scope, different open-source and enterprise providers proposed different ”linking services“, that have been called FaaS orchestrators. These are of course valid products but do not trust a standard, are not integrable and each of them does not absolve at all to all the requirements a functioning workflow could ask.

In this scenario, we started from scratch, defining the workflow concept. Therefore, we first determined what the elements involved in workflows, i.e., jobs, and events, and how they are related together. After that, we defined a design schema for workflows with clear terms, figures, and data models. Finally, using these tools, we proposed a reference architecture for the management of a workflow platform over a cluster.

This work can be considered a starting point for the serverless workflow field, but we still have to deal with different challenges, like i) designing the security aspects of the engine, ii) designing the fault tolerance aspects on the nodes, iii) implementing a workflow engine able to respect this reference architecture. All these challenges will be faced in the remaining chapters of this thesis.

CHAPTER 4

Deploying Continuum Native Applications

From our introduction, we understood that it is still impossible to build FaaS native applications without a Cloud broker that coordinates the functions on the continuum. Therefore, FaaS usage is limited to very simple and specific jobs. In this chapter, we brush up on Scientific Workflow using the FaaS paradigm to realize full Native Serverless Workflows-based applications.

We will face this challenge by designing two different platforms. The first is discussed in Section 4.1. In this work, we define a custom Workflow Manifest DSL used to describe function interactions; then, we describe the implementation of an agent able to deploy architecture-independent functions and coordinate them according to the Manifest. Finally, federating the Cloud-Fog-Edge tiers in a single Continuum environment, we allow functions to take advantage of the Continuum tier's characteristics where they are deployed. This project is called OpenWolf, and its repository is published on GitHub, under GNU General Public License v3.0, and is based on the reference WES architecture we described in Chapter 3.

The second approach we used to define continuum native distributed workflow is discussed in Section 4.2. In this work, we generate dynamic workflows using a distributed broker able to match IoT producers and Serverless Functions producers and consumers by the use of a profile data rule engine. This project enhances an already existing research project called RPulsar, and improves it letting it spread functions instead of no-scalable pieces of code in the Continuum.

4.1 OpenWolf: a Serverless Workflow Engine for Native Cloud-Edge Continuum

Recently, many works have been raised with the aim of enabling continuum computing, namely the ability to connect and orchestrate by events data computation between Cloud, Fog, and Edge for keeping a good QoS, i.e., network latency, computational power, and data locality. Many interesting solutions were proposed to solve the Continuum problem using proactive or reactive approaches. The Serverless paradigm has also established itself as a potential solution for designing continuum native applications, and in Chapter 3 we have already defined a reference architecture based on the Faas to deploy workflows on the continuum.

In this Chapter, using the concepts inherited from Chapter 3, we make use of the Serverless paradigm to (i) enable Faas over the continuum tiers, and (ii) solve the problem of composing complex functions-based architecture proposing a many-to-many workflow engine. In the following, we define the design, development, and validation of an Open Source Serverless Workflows Engine called OpenWolf, able to compose functions among the constrained Continuum tiers according to a workflow manifest. Source code has been published on GitHub[1] under GNU GENERAL PUBLIC license.

4.1.1 State of the Art

In Chapters 2 and 3 we have already argued the challenges of the Continuum, that is the ability to bring the computation at any infrastructure level (i.e. cloud, fog, and edge), and we have seen as the Serverless can help on this aim. Building Faas-based applications means building workflows, but as pointed out in [35], the three principles of serverless are (i) black-box functions, (ii) substitution, and (iii) double billing, cannot be respected in functions workflows. Even in these constraints, interesting works [33] and [53] have emerged with the scope of moving Cloud workloads as close as possible to the Edge using FaaS composition. However, the computation is focused on a use case and does not allow for building a general-purpose serverless composition model.

4.1.2 Motivation

A good opportunity for developing Continuum native applications is given by Serverless and FaaS platforms, which allow developers to focus on business logic applications (i.e.,

[1] https://github.com/christiansicari/Open-Wolf-Serverless-Workflow

function), demanding building, deployment, and API exposition to the platform itself. This seems to be a way to enable the continuum, but some challenges still need to be solved. First of all, Serverless does not guarantee architecture transparency. Indeed, it is not possible to transparently deploy functions over x64 platforms in Cloud and Fog, while ARM architectures are used in Edge.

Secondly, general-purpose applications are composed of several tasks related to different relationships, such as (i) following, (ii) dependency, and (iii) mutual exclusion. Therefore, they can be strictly sequential or parallel.

Recently, many projects aimed to implement FaaS workflow using open-source projects. However, the repositories are unmaintained or rarely updated, such as [54], [55]. Other projects, such as OpenWhisk, allow only the implementation of sequential function chains that are not enough for a general-purpose Workflow execution.

Using the principles and concepts defined in Chapter 3, this work aims then to exploit the Scientific Workflows principles when applied to Faas, to make the continuum happen. To achieve this, we outlined a roadmap as follows:

- defining a workflow manifest model to describe the workflow structure both in terms of involved processes, complex process relationships (one-to-many, many-to-one), used functions, and Continuum constraints;
- federating the Continuum computing layers in a single (high-level) homogeneous cluster built with heterogeneous computers/devices;
- providing the cluster with a Function Provider able to deploy and invoke function at any Continuum layer, considering also deployment constraints;
- building platform-independent functions usable at any Continuum's tier;
- enhancing FaaS platform with a composition engine able to (i) trigger workflows, and (ii) orchestrate the functions outputs' flow to the correct next function(s) in the workflow manifest.

4.1.3 OpenWolf Engine

In this section, we present the OpenWolf engine, an open-source project developed with the purpose of defining a new way for bringing computation at any tier of the Continuum, orchestrating and composing FaaS for implementing complex applications workflows.

The Serverless Workflow

As widely explained, a scientific workflow is a composition of processes and data that shape more complex applications. FaaS acts as scientific workflows' building blocks, but we have to manage functions to design workflows that can:

- support single/multiple trigger(s);
- support many to many relationships between functions;
- support data pre- and post-processing filters;
- support parallel and sequential process execution;

To the best of our knowledge, the most used open-source Serverless engines (i.e. OpenFaaS and OpenWhisk) do not natively realize such workflows, allowing only sequential composition of functions without data pre-and post-processing, as shown in Figure 4.1.

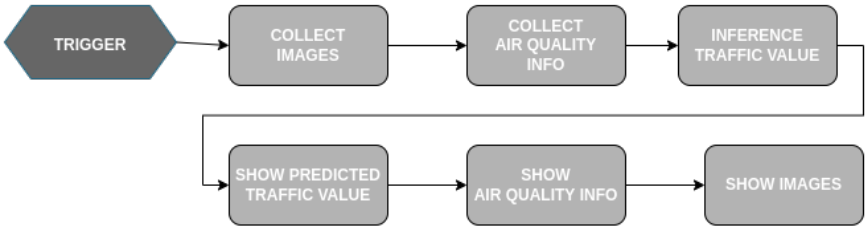


Figure 4.1: Sequential Function composition workflow

Instead, OpenWolf is designed to match all the above-mentioned properties, drawing workflows like in Figure 4.2. Figures 4.1 and 4.2 suppose a simplified road traffic inference based both on collected images and information related to the air quality. The difference is significant at a glance because in Figure 4.2 a single function can either trigger multiple ones (one-to-many) or wait to receive inputs from one or more previous functions (many-to-one). In these examples, the key difference is given by the functions that show data. In the first case, this waits for non-related functions, whereas, in the latter case, this fires once their inputs are ready ad then respects the real task dependency. This helps avoid any non-required waiting time.

Workflow Manifest

Typically, we need to use a Domain Specific Language (DSL) to describe a domain-specific structure. However, considering the young age of serverless workflow, any DSL

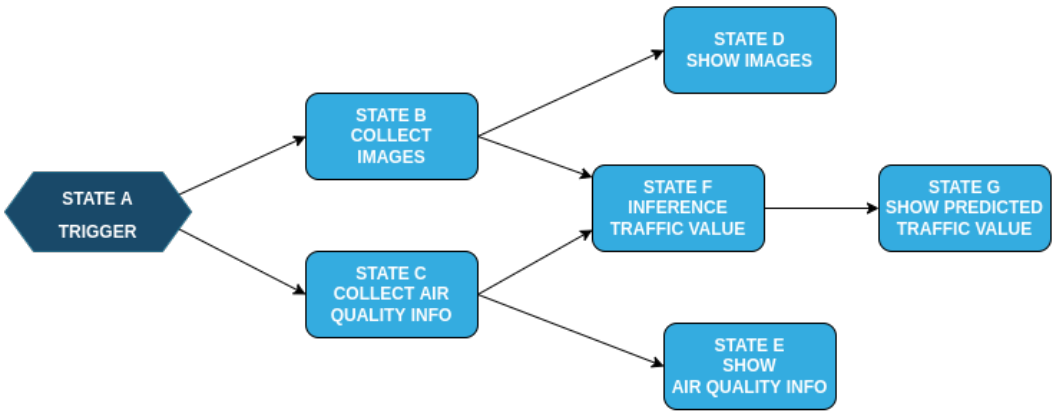


Figure 4.2: Example of Workflow in data analysis

has been established as a standard so far. A very interesting project in this field is given by the Cloud Native Computing Foundation (CNCF), which is currently maintaining the Serverless Workflow Project[2]. This project proposes a vendor-neutral, open-source, and fully community-driven ecosystem for defining and running DSL-based workflows that target the Serverless technology domain. Inspired by this, we defined a custom workflow manifest composed of three parts and reported its structure in the listing 4.1.

```
name: <workflow-name>
callbackUrl: <uri-where-to-send-result>
states:
  <state-id>:
    function:
      ref: <ref-to-function-id>
      config:
        key: value
      constrains:
         key: value
    start: true
functions:
  <function-id>:
    platform: openFaaS
    endpoint: <endpoint-to-function>
    config:
      key: value
workflow:
  <state-id>:
    activation: <Boolean Equation>
    inputFilter: <jq command>
```

[2]https://serverlessworkflow.io/

```
22        outputFilter: <jq command>
```

Listing 4.1: Workflow Manifest format in OpenWolf

The *functions* part contains all the function definitions that are consumed in the workflow. In each definition, we need to indicate at which URL it can be triggered and the default configuration parameters. The URL is provided by the Serverless provider (in this case, only OpenFaaS) when we deploy a function, the parameters instead depend on the function business logic, OpenFaaS typically allows us to send them using the HTTP headers that will be translated in environment variables.

Furthermore, we can add scheduling constraints to the function, a very useful property for selecting the right Continuum tier where the function will live.

The *states* part contains the set of the workflow's building blocks. They are used to envelop a function and can be triggered only once for each execution. For each state, we need to specify which function is triggered when the state is active and the configuration of custom parameters that override the default ones previously declared as a function parameter.

The *workflow* part defines the rules that link the states together. For each state that composes the workflow, we need to give the action function, which is a boolean combination of the other states, and when it is verified it tells the OpenWolf agent to trigger the State. Moreover, each state output could be not compatible with the input of the next State in the workflow, for this reason, we need to filter them. We do that using the inputFilter and outputFilter properties, enhanced with JQ instructions. JQ is a command-line utility that is thought to extract information like grep or awk, but it is specialized on JSON documents.

In summary, the manifest shapes the state interconnections, state dependencies, and their deployment over the Continuum. Therefore, we visualize the workflow as a dependency graph as designed in Figure 4.3.

Event Data Structure

The output returned by the state/function when this terminates its execution is defined within the event data structure. This is expressed using a JSON format, in which the main properties are called *ctx* and *data*.

The *ctx* represents the event context and it is composed of the *workflowID*, which references the workflow to which event it belongs, the *execID*, which distinguishes the different executions of the same workflow, and the *state*, which references the state that has returned the event.

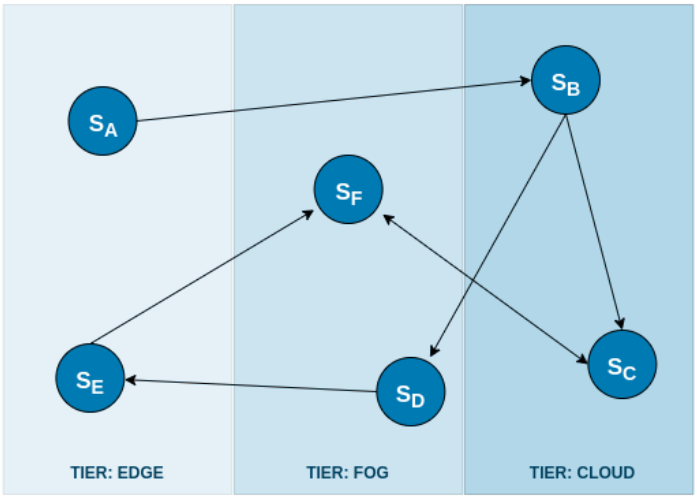


Figure 4.3: Workflow state graph in the Continuum

The *data* property, instead, is the function's output itself and, unlike the *ctx* that is read and set by the workflow agent (we will see it later), the function fully manages this.

An event example is proposed in the listing 4.2, which is fired by State C in the workflow shown in Figure 4.2.

```
1
2  {
3      "ctx": {
4          "workflowID": "inference-traffic",
5          "execID": "inference-traffic.123",
6          "state": "C"
7      },
8      "data": {
9          "AIQ": 47,
10         "Scale": "EU"
11     }
12 }
```

Listing 4.2: Workflow Event data model

OpenWolf Architecture

From the architectural point of view, we have to federate the Continuum layers, that is, creating a single computing cluster that collects every node in the Continuum and manages them with the same interface. This process allows the orchestrate and composition of the functions, as well as the spread of them in the Continuum. For federating the Continuum, we used K3S, a Kubernetes distribution mainly thought to be executed in unattended, resource-

constrained, remote locations or inside IoT appliances, even maintaining all the Kubernetes features. K3S is mainly used in Edge environments, in fact, it can be run in ARM64 and ARMv7 architectures, but it also supports x64 platforms. Indeed, this requires very accessible system characteristics with only 1 GB of RAM and 1 CPU installed. K3S can then be easily used to federate a Continuum environment by installing an agent in each node of the Continuum.

Over K3S nodes, we have to install a Serverless Engine to build, deploy and trigger functions. We chose to use OpenFaaS because of the capabilities to (i) build functions for multiple architectures at one time, using Docker's Codex, (ii) integrate with Kubernetes, (iii) natively use Kubernetes features such as node selectors, affinity, and anti-affinity for conditioning the function schedule, and (iv) support synchronous and asynchronous function invocations. These features allow us to develop a single function and make it automatically suitable for deployment at any Continuum tier.

The cluster architecture is also provided with a Redis instance that is used to store the workflow manifests and the workflow execution's information. Finally, the OpenWolf Agent, from inside the K3S cluster, works as a bridge between the components, coordinating them and the workflows' functions.

The overall architecture is finally shown in Figure 4.4.

The OpenWolf Agent

To achieve the composition, OpenWolf has to ensure that any event will follow the correct path in the workflow it belongs to and then trigger the correct states in the workflow with a proper transformation of the right income event. The OpenWolf agent is deployed as a standalone stateless microservice inside the Kubernetes Cluster we used to run the serverless functions.

The Agent exposes two interfaces. The first one is a public interface used to trigger a workflow from the external. The second one is closed inside the Kubernetes cluster and it is used as a callback URL for each asynchronous function triggered by any workflow. By doing that, the agent intercepts all the events belonging to a workflow, extracts the context information, and uses it for fetching all the workflow and current execution information. Therefore, it triggers the next states in the manifest, forwarding the right received event with the updated *ctx* property. This process is described more concisely in the activity diagram in Figure 4.5.

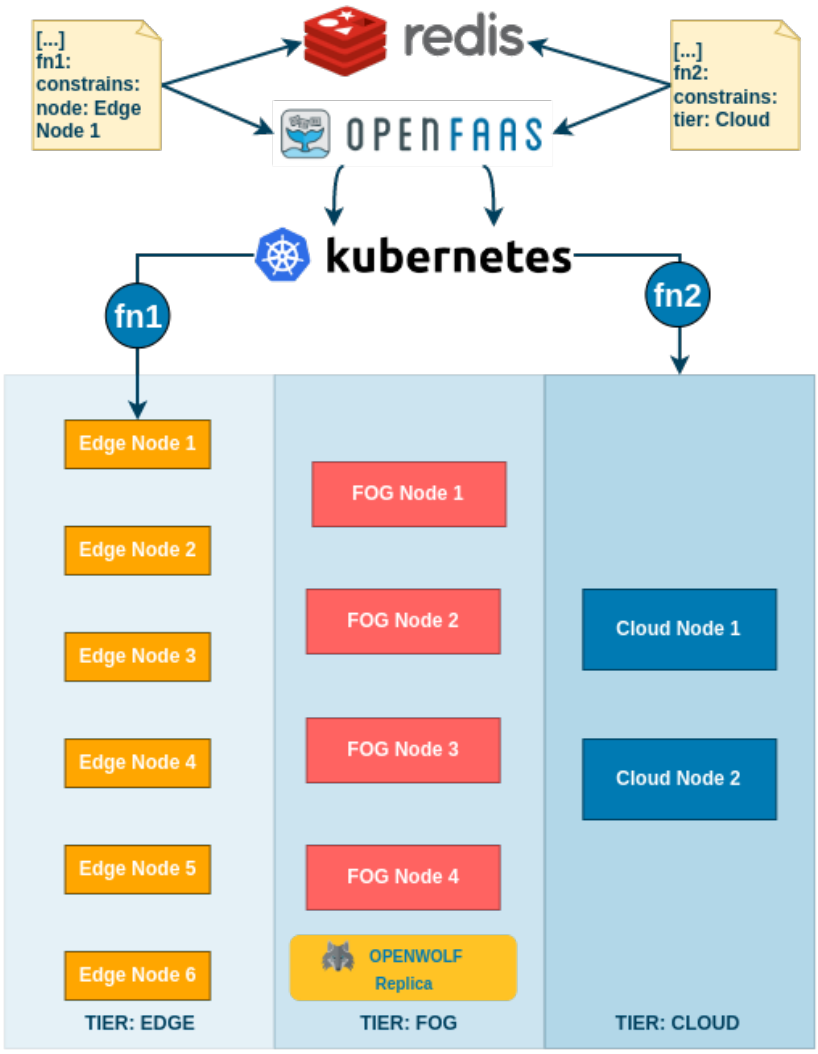


Figure 4.4: OpenWolf architecture

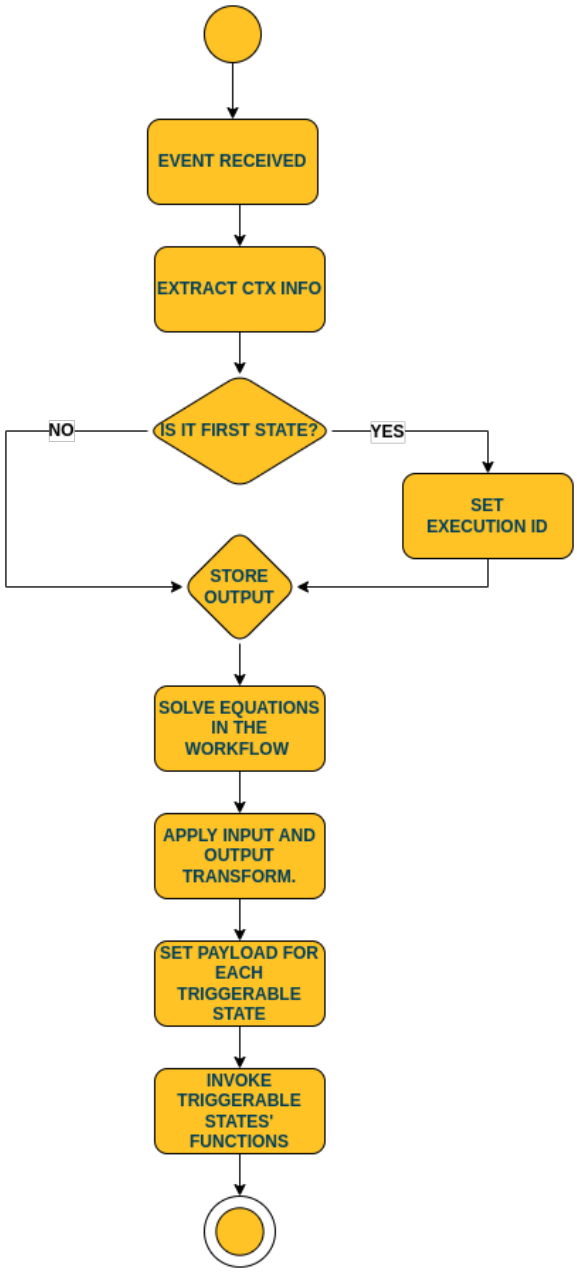


Figure 4.5: OpenWolf agent activity diagram

4.1.4 Performances

In this section, we present the results obtained while validating the architecture shown in Section 4.1.3. Our tests focused on five different challenges:

1. **Asynchronous vs Synchronous Calls**: OpenFaaS and many other Serverless platforms give the possibility to call a function in the foreground, waiting for the response (synchronous), or in the background, receiving the response using a callback (asynchronous). For scalability and resource usage reasons, OpenWolf needs to use only asynchronous calls, then we need to ensure that they guarantee comparable performances with the synchronous versions.

2. **Low latency during the chaining**: Typically FaaS runs standalone, whereas OpenWolf gives the possibility to organize functions in complex workflows. To achieve this, we need the agent to intercept the functions' output and recursively trigger new functions. This process requires a few steps that could decrease the overall system performance. Therefore, we want to understand what delay the agent adds to the system.

3. **Linear dependency between states number and execution time**: OpenWolf is thought to execute general-purpose workflows, this means that the number of the state to triggers could vary. Therefore, we want to ensure that increasing the workflow's state number, OpenWolf does not collapse, that is guaranteeing a linear execution time.

4. **Parallelization Capacity**: Unlike classical FaaS providers, OpenWolf is able to simultaneously trigger many functions at once, then join their outputs in a unique function. This kind of parallelization should improve the overall system performance, then we are interested in comparing a sequential behavior versus a parallel one.

5. **Scalability at the Continuum**: OpenWolf gives the possibility to run multiple functions at the Continuum, building complex Continuum Native Serverless Applications. This feature can be used only if the Fog and the Edge do not represent a bottleneck for the entire application, therefore we will go to test the same workflows run in a full Cloud/full Edge environment and in the Continuum, in order to compare their behaviors.

System Testbed

OpenWolf has been tested using a five-node Kubernetes cluster, composed of two nodes in the Cloud tier, one node in the Fog tier, and two nodes in the Edge tier. In the Cloud tier,

we run two different machines in an Openstack environment, where we run OpenFaaS's Gateway, Prometheus, Authentication Server, Kubernetes Master, and a Redis instance. In the Fog, we used a single workstation, where we run both the OpenWolf 's Agent, OpenFaaS' Nats, and Queue Manager. All these components are in charge of exchanging data between Cloud and Edge, sending and delivering messages to functions and agents, for this reason, we put them in the middle between the Cloud and the Edge tiers. Finally, we used the Edge tier to host only the OpenFaaS' edge functions deployed in the workflow. The systems' characteristics are summarised in the table 4.1. All tests reported in the following are based on a workflow composed of custom Hash functions, that once receive a data payload return the input's SHA256. About the platform, OpenFaaS can be configured with many parameters to have the best performance considering many environmental constraints. Our one is the result of many tests carried out with the scope to get an optimal configuration for any tier, and we reported it in table 4.2.

Instances	**Tier**	**Model**	**CPU**	**Memory**	**Operating System**
2	Cloud	Openstack VM	Intel Xeon 4.0 GHz, 8-core	32 GB	Debian 11
1	Fog	Workstation	Intel i7 4.4 GHz, 4-core	8 GB	Ubuntu 20
2	Edge	Raspberry Pi 4	ARM64 SoC 1.5GHz, 4-core	4 GB	Raspberry OS ARM64

Table 4.1: Cluster's nodes characteristics

Parameter	**Value**	**Condition**
Queue Workers	1	Ever
Function replicas	State references	States Number < 60
Function replicas	(State References)/2	States Number ≥ 60
Max_inflight	Equal to functions replicas	Ever

Table 4.2: OpenFaaS and OpenWolf parameters for the tesbed

Results

The first test we carried out aims to compare the synchronous and asynchronous execution time of Hash functions by using a webhook URL and an internal cluster logger. The webhook URL is a typical OpenFaaS concept, when a function is called asynchronously, its result is sent as POST payload to the webhook URL. We repeated this test by running the same

function in the Cloud and Edge tiers, executing 30 tests for each tier. The results are shown

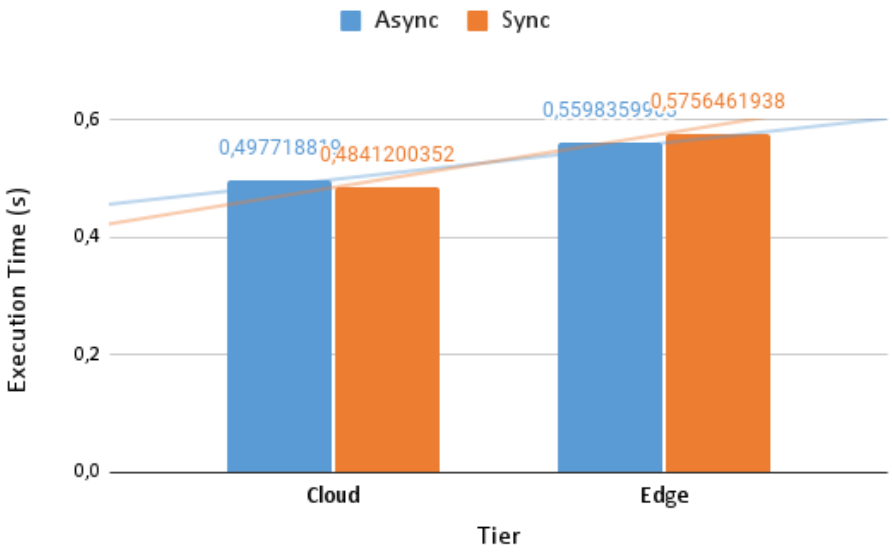


Figure 4.6: Synchronous vs asynchronous function execution time in OpenFaaS

in Figure 4.6. There is a really small gap between the synchronous and the asynchronous calls, which is under 2%. This is true for both tiers, with worse performances on the Edge one, as could be forecast. These results highlight that asynchronous function calls do not degrade performances, both considering deployment on Cloud and Edge. Moreover, Edge functions require more time than Cloud functions, however, given that the difference in terms of execution times is small, this does not generate bottlenecks in the workflow.

As often said, OpenFaaS allows running sequential workflows, composed of only two functions. In contrast, OpenWolf can be used to sequentially chain an undefined number of functions, overcoming the OpenFaaS' limits. Therefore, in the next test, we tried to design multiple sequential workflows in order to verify the scalability guaranteed by OpenWolf . We measured the time needed to execute an entire sequential workflow with an increasing number of states. In this case, the bottleneck might be given by the agent that might not manage to consume and deliver all the events in time.

As we see in Figure 4.7, we tested OpenWolf using nine different workflows composed of 3 to 90 states, increasing by 10 states at a time. As wished, the workflow's execution times increase linearly with the number of states, the ratio of states/execution time has an average value of 1,29Hz and a maximum value of 1,59Hz. These tests confirm a clear linear dependency between the number of the states and generally a quite low latency during the sequential function chaining.

The next test measures how much time we could save if we would have triggered multiple

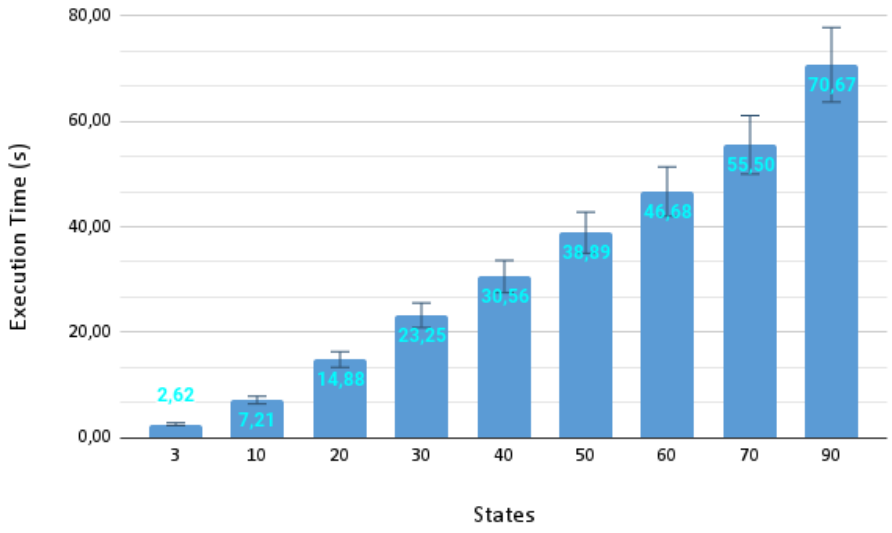


Figure 4.7: Sequential workflow execution time in OpenWolf

states at once, instead of running them one by one. Therefore, we compared the execution time of running the same workflow's states both in sequence and parallel. The results are shown in Figure 4.8. The tests have been carried out considering an increasing number of workflow states, from 3 to 90 and the FaaS parameters used are the same as reported in table 4.2. As shown in the Figure running the same workflow in parallel requires about 40% of the sequential time, then giving a great time-saving.

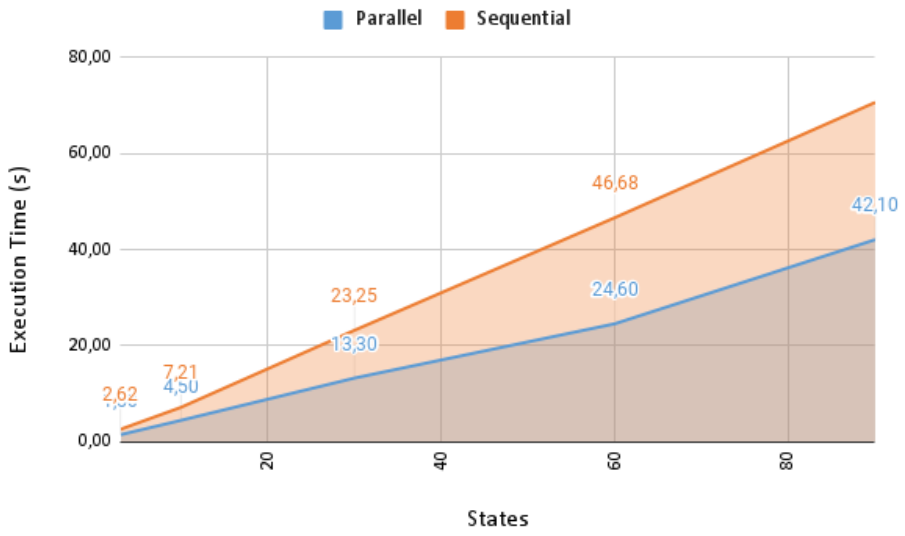


Figure 4.8: Sequential and parallel workflow execution time comparison in OpenWolf

Our last test compares the execution time of multiple parallel workflows when they are totally executed in Cloud, Edge, and the Continuum. We still used the configurations

reported in the table 4.2, but in the Continuum environment the functions have been equally spread among all the nodes, in all the tiers. The distribution among the tiers regards only the functions, OpenFaaS' components, and OpenWolf agent that have been deployed in the Cloud/Fog as said before.

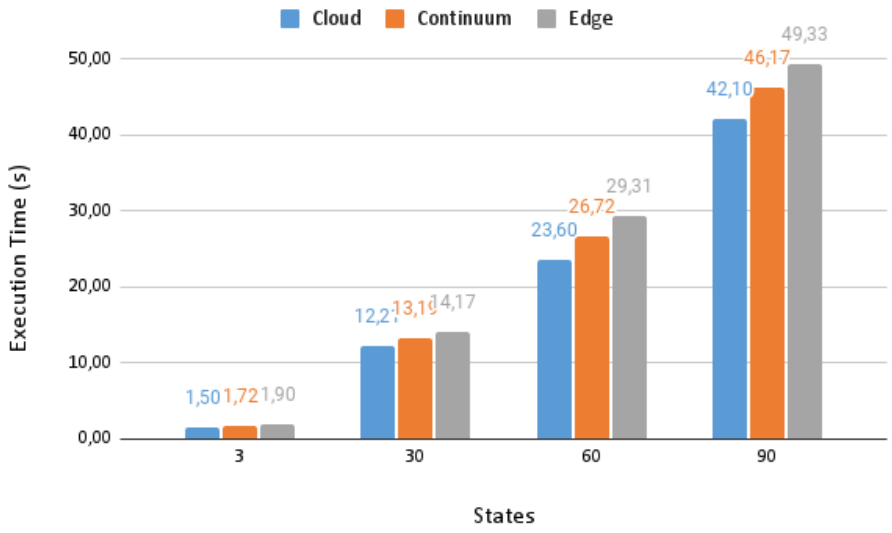


Figure 4.9: Workflow execution time in OpenWolf in different infrastructures

Performance differences in Cloud and Edge are given by two factors: (i) the computation capacity of the node that runs the function and (ii) the latency generated by the OpenWolf 's agent to contact the node that runs the function. The first factor is quite mitigated by the simplicity of the function we used in the test, the second factor is mitigated by the network topology we used, with the OpenWolf agent in the Fog, equally distant from the Cloud and the Edge. Given that, we expected comparable performance in any tier and in the Continuum. Figure 4.9 confirms that, in fact, even if a Cloud function guarantees better performances than an Edge, the percentage time increment is on average under 21%. In the Continuum instead, of using also the Fog as a computational tier, we get in general an execution time that is in the average between the Cloud, and the Edge, with a percentage time increase with respect to the Cloud around 15%. The deployment in the Continuum does not affect the execution time.

4.1.5 Conclusion

In this Chapter, we presented OpenWolf , a general-purpose Serverless Workflow Management System. We aimed to build a system able to exploit the FaaS paradigm for composing complex scientific workflows and connecting one or more functions distributed over the Cloud-Edge Continuum. We then federated a Continuum environment using a heteroge-

neous Kubernetes Cluster based on K3S, then we deployed over it OpenFaaS, a popular serverless platform, and we used it to build architecture-independent functions. Finally, we designed and developed a Workflow Agent, a small microservice able to parse Workflow Manifest files and then intercept and forward functions' data according to the workflow structure.

Our novelty is mainly given by the marriage between Serverless and Scientific Workflow that until now has been only mentioned but never documented. We made this possible by improving the actual serverless platforms giving the possibility to describe the workflow through a Manifest. Furthermore, we managed complex function graphs composed of one-to-one, many-to-one, and one-to-many relationships with the Workflow Agent, overcoming the actual open-source serverless platform limits.

4.2 Event-Driven FaaS Workflows for Enabling IoT Data Processing at the Cloud Edge Continuum

The Internet of Things (IoT) is one of the biggest data sources on the internet, and nowadays, it is used to serve many different use cases, such as smart cities and environment surveillance[56], sports [57], healthcare[58], and Industry 4.0 (IIoT) [59].

IoT data often requires real-time processing when the data is time-sensitive and requires immediate action; this means that data is processed as it is generated rather than stored for later analysis, but deciding what and when data coming from one or more sensors must be consumed in real-time or later can frequently vary depending on many factors like Quality of Service (QoS) constraints, human or consumer needs, or the data itself.

Starting from those constraints, the concept of Continuum Computing arose with the aim of taking advantage of the well-known cloud, fog, edge, and IoT infrastructures to run data analysis algorithms when they best fit at a specific moment [30], considering factors like network latency [60], energy [61], data location[62], and pipeline optimization [24]. Unfortunately, as highlighted in the scientific literature [10, 63], Continuum native apps do not exist, and we need to redesign everything that is supposed to be run on the Continuum.

At the moment Continuum Computing is affected by some problems, such as:

1. inability to dynamically react to an environment, application, or data change;
2. inability to keep an application stateless and therefore relocatable without data loss;

3. inability to profile the behavior of a specific task and then forecast its needs in the future.

Recently, R-Pulsar[50] has been introduced to address the first listed issue. This continuum-native framework, in fact, can federate IoT, edge, and cloud infrastructures, binding data and data consumers using an advanced profile match system and relocating both data and analysis using a dynamic rule engine system able to capture the change in the data and react accordingly.

To address the remaining two issues, we need to introduce and use Function as a Service (FaaS), as we already did in OpenWolf.

In this work, we aim to address the previously highlighted challenges in the field of Continuum Computing together. We do that using RPulsar to federate the infrastructure, orchestrate the computation, and react to some changes in the data or on the environment, but we upgrade it, providing a distributed Faas layer that lets to define, deploy, relocate, and analyze functions. Indeed, the contribution of this work consists of providing a framework to:

1. compute at the Continuum using FaaS functions;
2. monitor and analyze functions when executed on different environments with different data and configurations;
3. dynamically react to a change in the data, or in the environment, adapting and relocating functions on the continuum.

4.2.1 Use Case and Related Work

This research was driven by a typical scenario in Urgent Computing for IoT: detecting fires in vast and remote regions. The objective is to enable emergency services to swiftly and effectively identify areas that are safe, at risk of fire, or already engulfed in flames.

The initial step involves detecting the presence of smoke in the affected zones using air analyzer sensors. Subsequently, the collected data is preprocessed at the edge of the network, utilizing the computational resources available on board to analyze its content and determine if any additional postprocessing is necessary. If further processing is required, the data is transmitted to the cloud for change detection analysis. This analysis can involve comparing the data to historical records or simply for storage purposes.

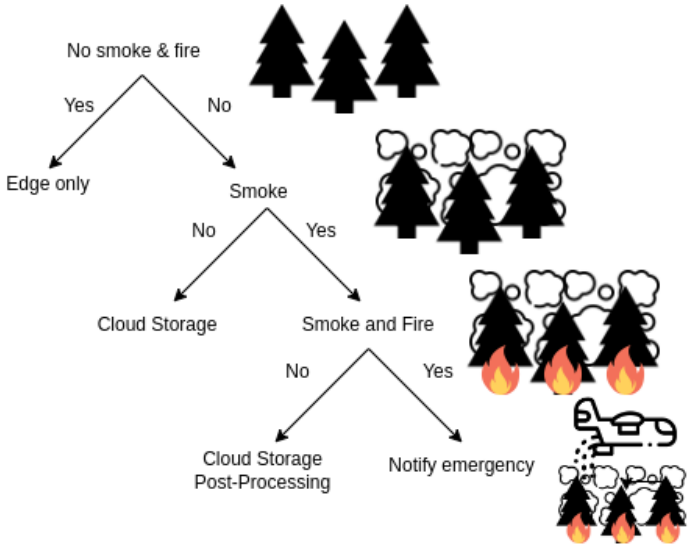


Figure 4.10: Urgent computing decision stage and reactions

Related Work

Many recent works addressed the continuum problem from an architectural point of view; in [64] proposed a data-driven model to publish and compute data everywhere, where basically the continuum problem is addressed statically by deploying a consumer in a specific node without having the chance to move around. A similar idea is proposed in [18] The authors propose a custom pub/sub broker with relocatable consumers, but even in that case, the data storing location does not have any optimization. In [65] authors tried to take advantage of Kubernetes to manage a cloud edge federation, and then by the use of a custom scheduler, they locate pods, trying to optimize the network latency between the pods that are supposed to communicate. This approach has become popular, but Kubernetes still force the continuum to be a centralized master-slave architecture, and the optimized parameters take care of the network bandwidth. In [66] authors propose a federated learning approach to schedule tasks from a task queue. Unfortunately, as highlighted by the authors themselves, knowing the nature of the task is not trivial, and predicting the best location can often be hard. Analyzing the history to find a better placement for a task is a strategy also adopted in [67] The authors here propose a scheduling algorithm based on the declaration of the needed data and the coupling degree with other tasks to try to optimize the request performances. Authors even keep a history of the previous execution and the target QoS, to better provide a kind of feedback to the next schedules.

This part of the State of the Art highlights the need to know the task to execute to understand how to deal with it. Unfortunately, generic process tasks are not easy to classify

unless they do not belong to a specific domain of tasks. Using the FaaS at this point would allow knowing the task by the function it is running, and this might help the task accounting. Using FaaS at the continuum is still considered a novelty, but recently some works with this aim have been proposed.

In [68] authors extended the work already discussed in [64], enhancing the Fiware-based infrastructure with Faas capabilities at the Fog layers. authors here easily profile functions but do not take advantage of that to better locate data or computation. In [69] authors still use Fiware stack to subscribe Faas platform ad at the edge of the network, but the replication of different Faas drops the bottleneck of having a single gateway, moreover, data are not scaled since they are duplicates in all the context brokers where they are needed. In [70] authors extended the work in [65], adapting the geo-scheduling used for Kubernetes's pods to OpenFaas' ones. This approach guarantees a reliable scheduler, but the presence of a Kubernetes cluster might be too stressful for constrained edge devices.

In [71] authors propose a data-declaration-based algorithm to schedule OpenWhisk-based functions; the idea of considering where the data are is good for knowing where to compute, but it doesn't guarantee any flexibility to the system. Furthermore, OpenWhisk is not well supported on constrained arm devices, and this might be problematic for edge computing [72]. Finally, the network-based scheduler has been used again, even for Faas scheduling in A [73] but this approach does not learn from the knowledge and just uses a static dataset to place a function at the edge better statically.

Finally, in [4], the authors propose to use an optimized Serverless Workflow engine built on Kubernetes to spread functions at any scale. In this project, functions can be scaled and deployed everywhere, as defined by the user, but still, Kubernetes might be hard to stand at any scale.

4.2.2 Background

RPulsar

The R-Pulsar system has been introduced with the purpose of gathering and analyzing data for applications that span both the cloud and the edge of the network. Its main goal is to extend cloud capabilities to edge devices, enabling them to collect and analyze data closer to the source and respond autonomously to local events. Initial investigations have focused on building upon and expanding the Associative Rendezvous (AR) interaction model, adapting it to support data-driven IoT applications. The AR paradigm facilitates

content-based decoupled interactions, where interactions are defined in terms of semantic profiles rather than names. These interactions offer programmable reactive behaviors, serving as the core services for data-driven workflow executions and decision-making. The disaster recovery use case was utilized to validate and evaluate these extensions, and the extended AR model was employed to support workflow topologies triggered and scheduled based on the content of data streams.

Considering that IoT applications require processing large volumes of streaming data through complex workflows in a timely manner, solely relying on cloud resources becomes impractical. Therefore, leveraging resources closer to the edge becomes crucial, although these resources are typically limited in capabilities. Consequently, it becomes necessary to balance the quality, immediacy, and cost of data processing in a context-aware manner. A rule-based system is utilized to address this challenge, where all relevant knowledge is encoded into a set of If-Then rules.

R-Pulsar adopts a distributed architecture using an overlay network, where each node within the overlay network is referred to as a Rendezvous Point (RP). The system comprises five layers: the location-aware self-organizing overlay, the content-based routing layer, the serverless messaging layer, the memory-mapped data processing layer, and the programming abstraction layer.

The location-aware self-organizing overlay, which incorporates location awareness, serves as an abstraction for the layers above it. With just one function exposed to the other layers, it is able to drive AR Messages to the closest RP to the data source.

The Content-based Routing Layer builds upon the location-aware overlay to facilitate message routing between clients of R-Pulsar. It leverages the underlying infrastructure to direct messages efficiently.

The Memory-mapped Streaming Analytics Pipeline is responsible for consolidating data from various sources, processing it, and making it accessible for further utilization.

The Serverless Messaging Layer enables the deployment and execution of code fragments as a reaction to some specific bound events without requiring the explicit specification of IP addresses.

The rule-based programming abstraction layer empowers the construction of IoT applications and the decision-making process regarding when data should be sent to the cloud for subsequent postprocessing. Importantly, it eliminates the need for developers to manage any underlying infrastructure.

Dynamic Faas scheduling

The weak points of a system such as OpenWolf that we described in Section 4.1, are the static assignment of a workflow function in a tier that is specified at the beginning of the manifest or otherwise decided by OpenWolf during the workflow bootstrap and the hardness of dynamically connecting new incoming functions to some previous ones. Unfortunately, as even highlighted in section 4.2.1, the urgency of a specific computation can vary according to the incoming new data, the system overloading, or because some external change, therefore being stuck on a function schedule, does not reflect the need of moving a function according to the urgency. Moreover, in reaction to some new data, the system may require to use of new functions in the same workflow, and this would require the application of a new workflow manifest, then wasting time, or even just changing the function invocation parameters.

In contrast, RPulsar natively supports both dynamic bindings of new producers and consumers and the possibility of reacting to some change in the data or in the system state. On the other hand, RPulsar does not support FaaS and, consequently, all the benefits it brings.

Rpulsar and OpenWolf are clearly complementary works; indeed, the aim of this work is to resolve the weak point of one with the strong point of the other.

To do that, we kept all the RPulsar core that lets us dynamically associate producers and consumers, but we changed the shape of a consumer that now is designed as a function wrapper, which contains the function reference it is supposed to run, as well as the function configuration that overwrites the default ones. At that point, the consumer can also act as a producer, which serves the function output as system data, and then more consumers can be bound with it. Doing that, a manifest is no more needed because the definition of a Workflow is implicitly done by the binding of producers and consumers. From RPulsar, we have even kept the rule engine, but we modified it in order to support the Function monitoring system, as well as the reactions that can be generated, for example, running a function in the edge instead of the cloud if the data, the required QoS, or the current system overloading change.

4.2.3 Architecture

Architecture Overview

The engine we designed in this work starts with the basic infrastructure that RPulsar already provides, but it also adds some peculiar components that are required to allow the spread and coordination of functions as well as the management of the data produced and consumed in a workflow. As we can see in Figure 4.12a, the architecture is composed of three

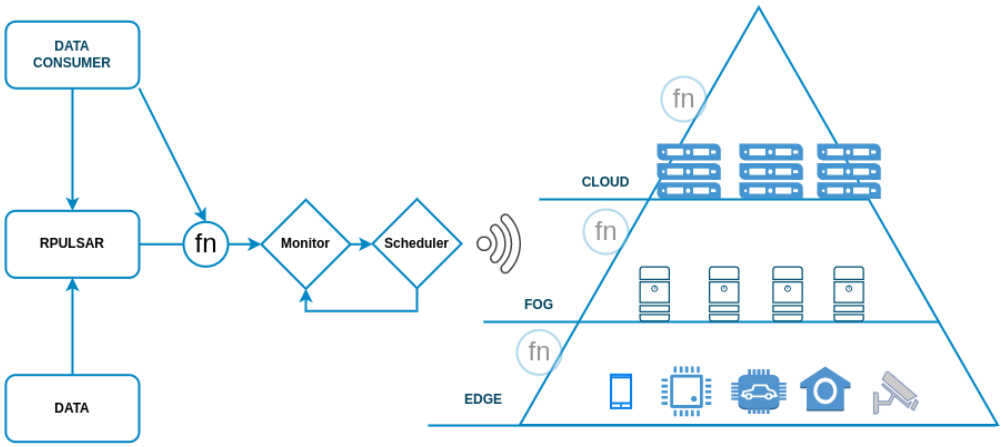
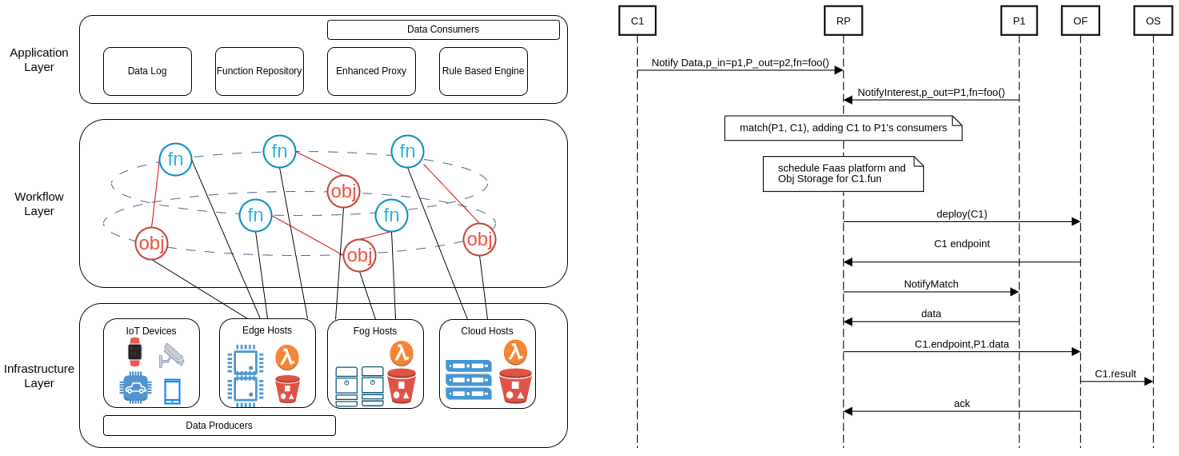


Figure 4.11: Function spreading on Continuum



(a) Architecture overview **(b)** ARMessage exchange sequence diagram

Figure 4.12: Enhanced Rpuslar architecture overview and peers' message exchange

layers: Infrastructure Layer, Workflow Layer, and Application Layer.

Infrastructure Layer The infrastructure layer includes all the hardware and primary services used in RPulsar, such as IoT devices and sensors used to capture and send data on RPulsar, and the Continuum infrastructure composed of edge, fog, and cloud nodes. Independent of their type, Continuum nodes are composed of a hardware part (embedded processors, workstations, and data centers) and a service part composed of a serverless platform and/or a storage area.
The **Serverless Platform** is the core of the project, and we implemented it using OpenFaaS. OpenFaaS is an open-source serverless platform that lets one build, deploy, and manage functions using Kubernetes or faasd. OpenFaas is composed of:

1. The gateway that lets us invoke functions or use API to interact with OpenFaas;
2. NATS that collect asynchronous function requests;
3. Prometheus which provides metrics about the running functions;

The **Storage Area** is realized using Minio, an S3-compatible object storage, and it plays a critical role in providing a space where functions can store and retrieve input and output data that, in turn, are used by all the functions in the triggered workflow. The Rule Engine at the Service Layer determines which storage area has to be used as well as where to run a function.

The Workflow Layer The Workflow Layer can be considered an abstract layer since it is not composed of components or hardware like the Infrastructure and the Applications layers, but it is shaped by the interaction of those layers. This layer is composed of the functions and the objects that are available during the life cycle of a workflow. Each function is represented using the *fn* symbol, and they are double linked with one (or more) object represented with the *obj* symbol, and both are connected to the Infrastructure Layer. These links exist because each object belongs to one or more functions; on the contrary, a function consumes and produces objects. This map between functions and objects is not hidden to the final users who submit data consumers to RPulsar. The Rule Engine determines the Workflow Layer and Infrastructure Layer links, which locates and relocates functions and objects to satisfy some given constraints. In this case, this mapping is hidden to the final users, who are not interested in where functions and data are located but in the satisfaction of the QoS parameters they

defined. Of course, functions interact with each other according to their profile matches, and this implicitly creates a FaaS workflow.

The Application Layer The Application Layer contains all the services needed to design, trigger, and adapt a workflow over the infrastructure layer, and it is composed of the Data Log, the Function Repository, the Enhanced Proxy, the Rule Based Engine, and the Data Consumers.

The **Data Log** is a distributed MongoDB instance, and each peer contains the sets of information related to the continuum host where they are installed. Each peer is filled with the information that the Enhanced Proxy can provide about the function's execution that happened in that node, and it can be queried to build metrics-based scheduling rules.

The **Function Repository** is a centralized service used to publish, version, and retrieve functions that can be used immediately on any node at the Infrastructure Layer. Functions are stored using pre-configured docker container images that allow building functions in Python, Java, Golang, Javascript, and Ruby. Each function can be cross-compiled and published for different architectures, like armv7, armv8, amd64, or ppc64le. The deployment of a function on a node will be successful if the right architecture image is available. Using a single Function Registry improves the code reusability, letting share functions among all the RPulsar users instead of using custom code at any time.

The **Enhanced Proxy** redirects the requests to a function to the Infrastructure node where that function is run for that specific workflow. In this way, we avoid directly interacting with a function, and then hiding the composition of the Infrastructure Layer. This proxy, while forwarding the requests and the responses to and from a function, asynchronously queries the Prometheus instance installed in the Serverless Platform where the function has been run and stores the fetched metrics in the Data Log to be used by the Rule Engine.

The **Rule-Based Engine** is the highest level service component that takes advantage of all the previous components to let the final users define rules that drive the position and relocation of a specific function inside a workflow. Those rules can be based on one or more of these factors:

- Incoming data;
- current loading of the system;
- History about previous similar jobs.

The policies can be evaluated at any new incoming data, then letting RPulsar dynamically spread and optimize the computation (and storage) on the Continuum when needed.

```
if(payload < 5){
    invokeFunction(faas=FZoneX, f=function, data=payload, cfg=config, zone_out=SX)
}else{
    invokeFunction(faas=FZoneY, f=function, data=payload, cfg=config, zone_out=SY)
}
```

Listing 4.3: RPulsar's rule based on payload

In the listing 4.3, we define a rule that selects a node where to run a function based just on the content of the payload. Depending on the value and the meaning of this value, we can define where to compute and where to store the function's output.

```
node=Datastore.getMetrics("TTR < 10s", "1d").sort("TTR: ASCENDING")[0]
invokeFunction(faas=node, f=function, data=payload, cfg=config, zone_out=node.closest())
```

Listing 4.4: RPulsar's rule based on system metrics

In the listing 4.4, instead, we are accessing the datastore to find the list of nodes where the Time to Respond (TTR) has been less than 10 seconds in the metrics collected in the last day, then we retrieve the first node from this list. The invocation of the function will then use the retrieved node to run the function, and it will store the function's output in the closest storage zone.

Message Exchange

RPulsar is able to build and activate Workflows using an advanced Pub/Sub model based on the matching of producers' and consumers' profiles. This process is deeply explained in [50], but we needed to modify them in order to include the use of the FaaS.

To start a workflow, we need producers and consumers to exchange ARMessages in the order shown in Figure 4.12b. Producers and Consumers do not know each other, then they just interact with an RPulsar node (RP), which will create a communication channel between them later. In the figure, the Consumer C1 sends a NotifyData message, This message is used to let RP know that it is interested in data that has a profile *p1*, and when it receives it, it will apply a *foo()* function on it, providing a result with a *p2* profile (C1 will become a producer

too then). Afterward, a producer (P1) sends a NotifyInterest message to RP, saying that it can send data with *p1* profile.

RP notices that C1 is interested in consuming P1's data; namely, there is a match. RP then activates the Rule Engine and schedules the C1's *foo()* function on one of the Serverless Platforms (OF) and the Object Storage (OS) to use in the Infrastructure Layer. RP will use the OpenFaas API to deploy the function on the scheduled node, it will receive back the endpoint to use to invoke the function. When the OF is ready, RP will send a NotifyMatch message to P1, then P1 will start to produce and send data. Finally, RP will invoke the function on OF sending the P1's data in the payload. Once executed, the function will store the result on the OS and inform RP that the computation is completed. If a new consumer C2 interested in data with profile *p2* will appear, this workflow restarts, but this time the producer will act from C1, instead of sending directly the data will tell RP the OS address where the data are.

4.2.4 Performance Analsyis

To test the improvement we have brought to this new version of RPulsar, we have set up a Continuum scenario where RPulsar is in charge of publishing and consuming IoT data using FaaS functions. In the scenario we considered, we aim at consuming sensor and camera data from the SAGE platform [3] to detect the presence of smoke. To publish sensor data, we will locate an RPulsar producer at the Edge of the network, while a second RPulsar node will consume it using a function that can be located in any FaaS Zone. In this testbed, we considered three zones located at the edge of the network where data are even produced, in the cloud, specifically in a High-Performance Data Center, and in one in the fog, namely in the middle of the route between the edge and cloud zones.

More details about those zones are given in the table 4.3.

Performance experiments

In this section, we have performed a set of experiments to compare the proposed processing approach with a traditional approach in which the stream processing is located in a fixed location at the core of the infrastructure.

We are interested in two metrics, the Computation Time (CT) and the Request Time (RT). The CT metric measures just the time to execute the function on the payload and get a result; it does not include any other task. The RT measures the time elapsed from when a

[3] https://sagecontinuum.org/

#	Tier	CPU	Memory	Operating System	Location	Network Upload Speed to Server	Network Download Speed from Server
1	Edge	Cortex-A72 4-cores 1.8 GHz	8 GB	Ubuntu 18.04 Aarch64	CloudLab Utah Datacenter	17.6 Gbits/sec	17.6 Gbits/sec
1	Fog	Intel i7-5930K 12-cores 3.50GHz,	32 GB	Ubuntu 22.04 Amd64	SCI Institute, Utah	543 Mbits/sec	542 Mbits/sec
1	Cloud	Intel Xeon 54-cores 2.00GHz	254 GB	Ubuntu 18.04 Amd64	Gigabit P2P CloudLab Clemson	407 Mbits/sec	406 Mbits/sec

Table 4.3: Cluster nodes characteristics for Enhanced RPulsar testbed

request is started to when the result is sent back to the client. Indeed, this metric involves the CT, as well as the network card's ability to compress and decompress the request, and the network bandwidth used to transfer the data from the client to the serverless platform. These two metrics are obtained from the Data Log component and can even be used to build performance-oriented deployment rules, as seen before.

We tested the Edge-Fog-Cloud Serverless environment, considering two main parameters: the payload size and the number of concurrent requests. Regarding the payload size, we have considered two different kinds of datasets, the lightweight, composed of five payloads of sizes 100KB, 200KB, 300KB, 400KB, and 500KB; and the medium weight, composed of five payloads of size 1MB, 1.5MB, 2MB, 2.5MB, and 3MB.

We selected these two datasets among all the possible choices because those are the smallest that allow us to demonstrate a change of behavior in terms of performances between the cloud, the fog, and the edge nodes. With respect to concurrent requests, we considered 11 different use cases, from 1 request at a time to 100. Even in this case, these values allow us to study a change of behavior in the computation layer.

The first simple test is shown in Figure 4.13. The test lets us see the difference in terms of hard computation on all three zones when they compute heavier payloads. As totally expected considering the resources described in table 4.3, the edge zone performs worse than cloud and fog zones for any payload size, and the gap between the zones increases, increasing the payload size by a super linear factor.

Fog and cloud zones, instead, keep comparable performance results for any payload size.

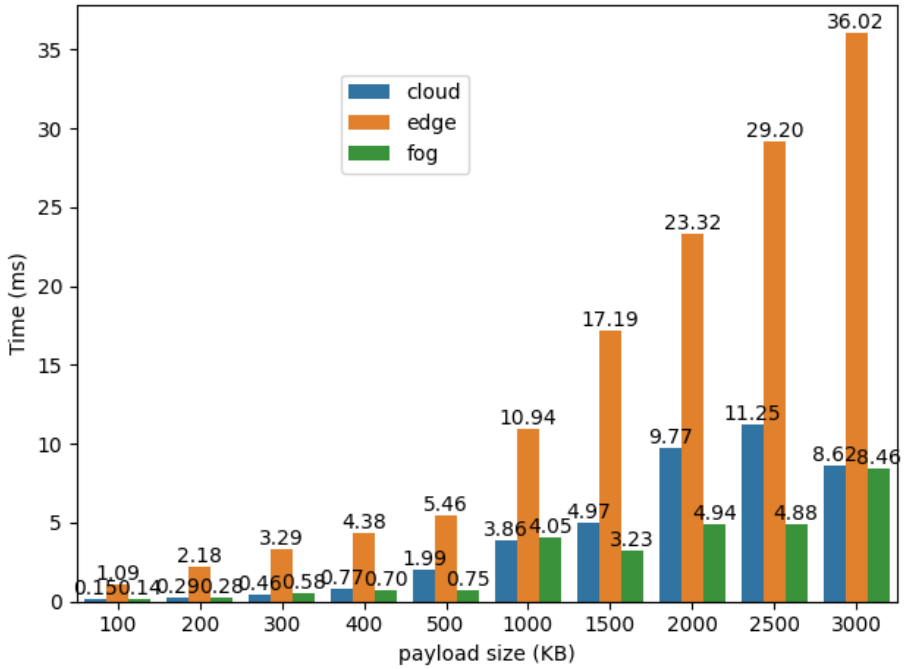


Figure 4.13: Best RPulsar' CT on continuum tiers increasing payload size

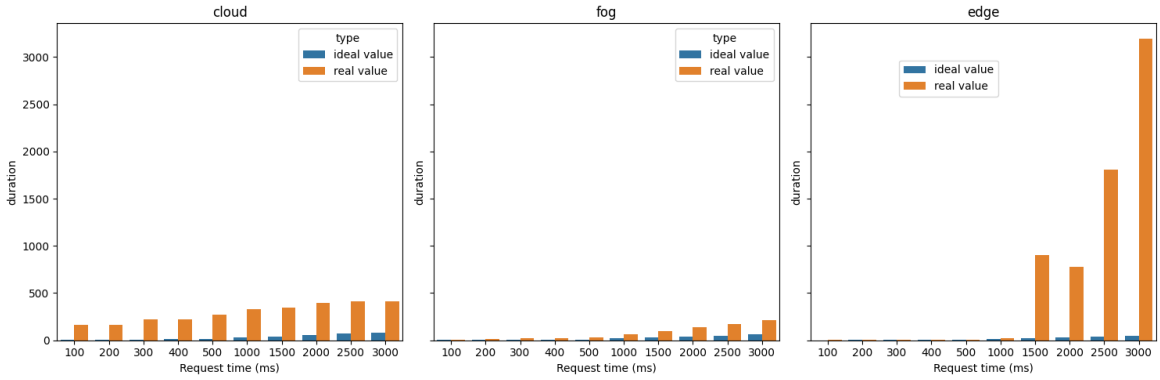
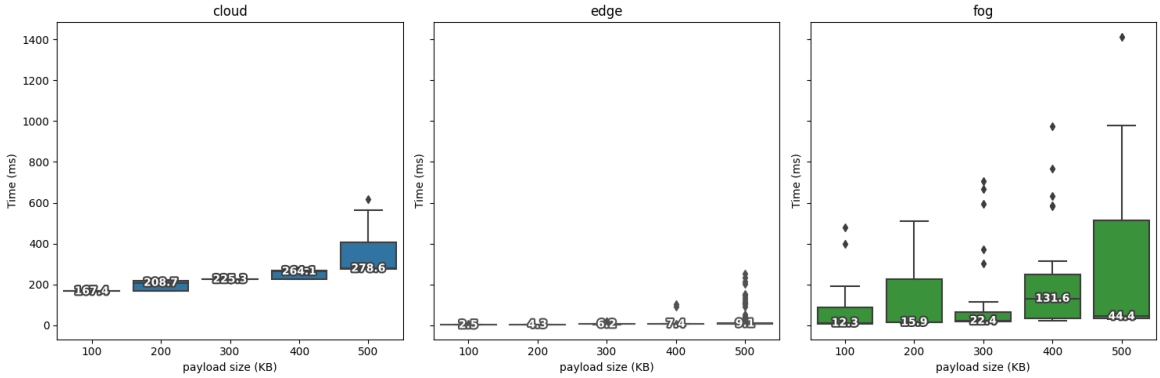
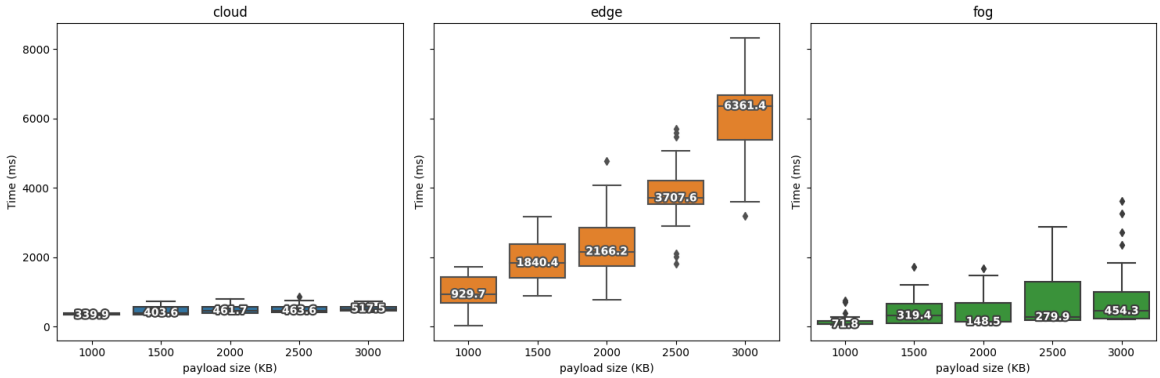


Figure 4.14: Ideal RPulsar request duration vs real one on increasing payload



(a) Enhanced RPulsar's RT in a single request with small dataset



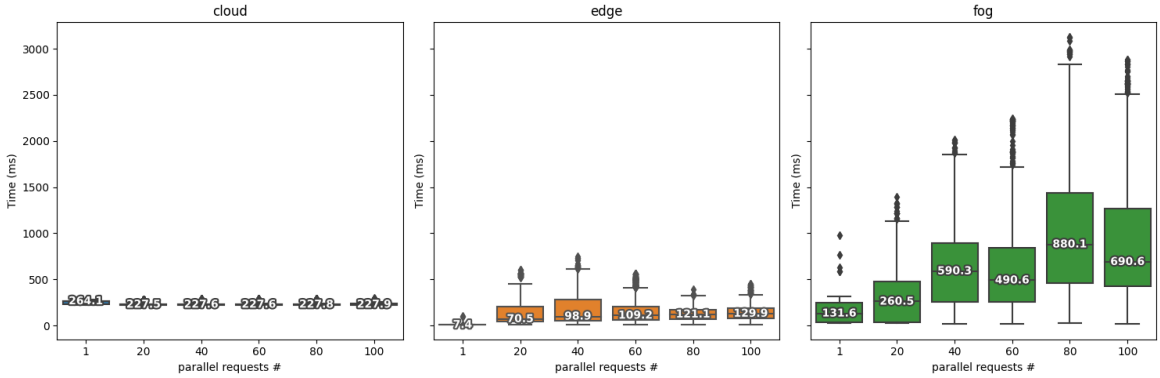
(b) Enhanced RPulsar's RT single request with medium dataset

Figure 4.15: RPulsar RT with one single request at time

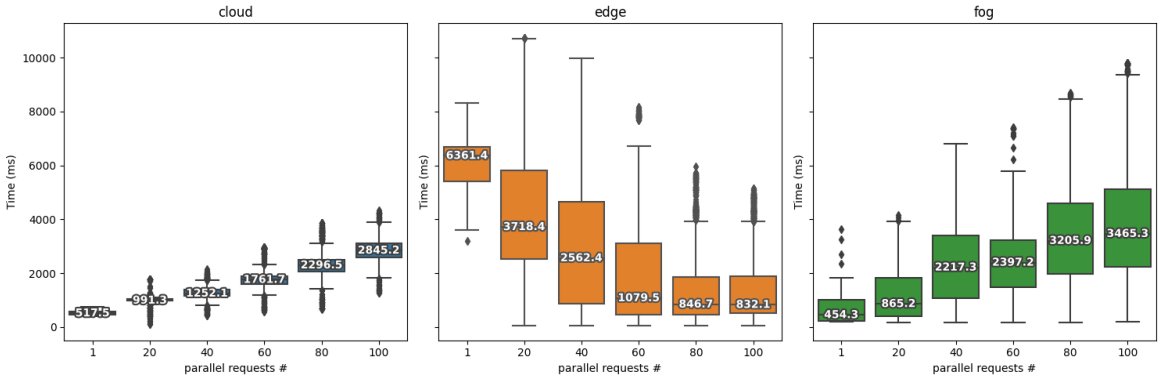
Using the data we collected using the results shown in figure 4.13, together with the network latency between the tiers that we have measured using iperf3 tool [4], and reported in table 4.3, we can estimate the ideal RT value even for different payloads size. Those ideal value has been compared with the real one, and the result is shown in figure 4.14. In the figure is clear that real performances are in general worse than ideal ones, in particular for cloud and fog, the delta between real and ideal is constant. That's not true in the edge where real and ideal performances are very close as long as the data computed are small, but as soon as we take into account the medium-weight datasets, the real value strongly decreases the performances, increasing the delta difference between it and the ideal value. This behavior lets us expect that the edge layer might in general perform better when the workload is light, especially because of the low use of the network, but it might rapidly change in the presence of heavier work, has we we are going to test soon.

In figure 4.15, we measured the RT, when just one request was sent per time on all three

[4]https://github.com/esnet/iperf



(a) Enhanced RPulsar's RT on increasing requests with 500KB payload



(b) Enhanced Rpulsar's RT on increasing requests with 3MB payload

Figure 4.16: Enhanced RPulsar's RT increasing parallel requests

tiers. Figure 4.15a shows the performance guaranteed using the small dataset. Here, despite the tests shown in figure 4.13, the edge dominates both the performance of cloud and fog; in fact, the involvement of the network that is needed to send the data away from the edge outweighs the time needed to compute the data, producing the shown result.

Subfigure 4.15b instead represents the same information in figure 4.15a, but using the medium-weight dataset. Results shown here totally revert to what was said previously; in fact, in the edge, we are no longer able to efficiently compute incoming requests, performing overall worse than both the cloud and the edge. Respect the previous figure, it is absolutely interesting even to analyze the stability at the edge; in fact, while for small payloads, all the tests run on the edge have almost the same performances, we see many more outliers and bigger interquartile ranges for bigger payloads.

Figure 4.16 wants to analyze the scaling capabilities on the three tiers, showing the RT when the payload size is fixed and the requests arrive with an increasing parallel factor.

In the subfigure 4.16a, the payload is fixed to 400KB; here, the cloud and edge highlight

both a stabler behavior than the fog. In particular, the cloud node does not significantly change its RT, while the edge node requires 16 times more to complete 100 requests than one. Despite that, all the executions are close to the median value, and outliers are rare. Fog node, instead, has the worst performance of both cloud and edge, but also, many executions are far away from the median value, making the expected value not reliable.

Finally, subfigure 4.16b analyzes the scaling property when the payload size is fixed to 3 MB. We have already demonstrated that the edge performs the worst under these conditions; in the presence of concurrent requests, this behavior is even accentuated. In fact, from the figure, we can see overall worse performance on edge at any scale, but also a high instability in the results, which can be derived by the huge interquartile ranges in the box plots. The stability of a result is a key point to be considered when some urgent computing is running. In this figure, for example, considering 100 parallel requests, the best median value we can achieve is in the edge that should give a response in 832 ms, against the 2845 ms in the cloud and 3465 in the fog. If we take into account even the outliers and the highest values in the boxplots, we can see that Edge and Fog might perform in the worst case much worse than Cloud. Due to that, if guaranteeing a result in a specific time range is critical, the reliability in the cloud is much higher. On the other hand, if we aim to respond as soon as possible, and the risk of not respecting this deadline is acceptable, the edge might serve this goal better.

4.2.5 Summary

In those tests, we aimed to measure the performances that a continuum environment can guarantee when provided with multiple independent serverless layers spread among the cloud and the edge. We have taken into account the scaling factor, measured as the ability to keep the same performances even in the presence of an increasing demand for computations or data to compute. Our results comply with the current state of the art in the continuum field; we have highlighted the advantage of using edge infrastructures to compute low demand and small computations, taking advantage of a near-zero network cost that balances just sufficient computation powers. Cloud instead, as expected, has highlighted better performances for high demand and heavy requests. All of those data are available in the Data Log, and therefore they can be used to write proper scheduling auto-balanced rules.

4.2.6 Conclusion

In previous works, RPulsar has been presented as an edge native tool used to design and deploy IoT-centric event-driven workflow through a powerful profile match engine that automatically connects generic data producers and consumers. Afterward, following the prominent rise of the FaaS open-source platforms, we tried to exploit them to take advantage of them to design and deploy continuum native workflows connecting FaaS functions by the use of a distributed broker called OpenWolf.

In this work, our aim was to adapt the flexibility of RPulsar to implicit design workflows to the OpenWolf potentiality to spread and reuse functions at any Continuum tier. We did that using RPulsar baseline to build a connected distributed Continuum environment; then we modified the producers and consumers' profile structure to provide even a function reference used to publish as well as consume data. Unlike any generic tasks, the FaaS functions shape a clear task domain, and foreseeing their behavior is easier if there is a function execution history to query. Because of that, we even improved the RPulsar Rule Engine to use the execution history to choose where to run a function in order to ensure a given QoS.

Once those updates have been realized, we deeply tested and discussed this new RPulsar version to measure the behavior of a continuum environment under different levels of system stress, using a smoking detection use case as a scenario.

CHAPTER 5

Guaranteeing Security and Privacy in Continuum Environments

This Chapter focuses on the security issues that affect the Continuum Native applications. Previously, we demonstrated that serverless workflows are a suitable solution for deploying continuum native applications. Thanks to a definition of standard architecture, private and public cloud, and edge infrastructure can cooperate to build very performing infrastructures. With OpenWolf, we have carried out the first open-source project to enable Continuum Computing, but many challenges related to different security aspects have not been discussed, like unsafe communication channels, the privatization of the data, or the injection of malicious code. In subsection 5.1, we address all those problems and even others, applying the already discussed Osmotic Computing principles to the OpenWolf's workflows, finally realizing secure by design continuum native computation workflows. Once those updates, we will validate again OpenWolf, even demonstrating a performance improvement with respect to the previous version.

Following, we will address the Authorization and Authentication issues in highly unstable but also dynamic Continuum environments. In subsection 5.2 we will find a solution to authenticate, authorize, and propagate dynamic access policy to workflow applications, keeping soft consistency between Identity Manager's (IDm) peers connected in an unstable Continuum infrastructure. This component will be tested in two different smart environment scenarios.

5.1 Secure-by-Design Serverless Workflows on the Edge-Cloud Continuum Through the Osmotic Computing Paradigm

OpenWolf [2] has been presented as the first open-source serverless engine capable of composing functions using three main components: a workflow manifest, a Kubernetes heterogeneous cluster, and a Broker Agent. OpenWolf aims to bring the serverless at the Continuum layer, but this leads to some problems related to the environment's security, such as the definition of a secure overlay network for federating both the Continuum nodes, storage, and transmission of sensitive data. These contents have been treated in the field of distributed computing, but to the best of our knowledge, no one argued it for a serverless Continuum environment.

A universal approach for dealing with these aspects was proposed in the Osmotic Computing paradigm [11], that we deeply described in the Chapter 2. We believe that OpenWolf is a good starting point for the development of a Cloud-Edge continuum system because it easily manages distributed function-based applications spread over Cloud and Edge. Nevertheless, as well as in many other Continuum computing engines, it raises some security threats that, in this work, we aim to solve through the Osmotic Computing paradigm. In this chapter we aim to address the following security aspects:

O1 **Secure storage for sensitive data**, which assures that confidential information is not disclosed to unauthorized individuals.

O2 **Secure and authorized secret distribution**, because serverless applications are distributed across the Cloud-Edge Continuum, and, in a so complex environment, defining a way for distributing secrets is crucial.

O3 **Functions access control**, which guarantees that the interaction between functions is under control, disallowing malicious functions can inject forbidden invocations.

O4 **Communication security** between nodes that prevents unauthorized parties from gaining understandable access to the communication, granting that the object of the communication is delivered to the expected receiver. This is also a complex security aspect in a serverless environment because serverless functions, which are spread across the Edge-Cloud Continuum, are deployed by a serverless function provider. Indeed, serverless workflows, composed of serverless functions, imply many internode communications, and it is impossible to a priori know on which particular node the serverless functions will be deployed or executed by the serverless functions provider.

O5 **Message security**, that consists in the process of isolating any communication between the components of an application, mainly through the encryption of messages of the exchanged data. In the case of a serverless environment, the entities exchanging messages are serverless functions that, by definition, are unaware of being part of a particular application. For this reason, guaranteeing message encryption is challenging.

The main scientific contributions of this research are:

- identifying security vulnerabilities of the existing OpenWolf implementation;
- implementing the Osmotic SDMem concepts to improve serverless security in the Cloud-Edge continuum;
- validate the implemented features with a non-secure implementation of OpenWolf, in terms of system execution time and resource utilization.

5.1.1 Related Works

When Serverless technologies are adopted, applications can be managed without the need to develop a server from scratch. By doing so, the service provider handles some security aspects. Recently, many works have been proposed addressing different aspects of the field of security in Serverless environments. In [74, 75, 76], authors underlined the fact that even if Serverless applications do not run on a managed server, they continue to execute code. If this code is not written securely, the application may be exposed to traditional application-level attacks. In particular, the security risks associated with serverless are summarized in ten points, such as i) code injection [77], ii) broken authentication, iii) sensitive data exposure, iv) XML external entities, v) broken access control [78], vi) security misconfigurations, vii) cross-site scripting, viii) insecure deserialization [79], ix) using components with known vulnerabilities, and x) insufficient logging and monitoring. In addition to that, it should be taken into account that applications are highly scalable in a serverless environment, and many parallel computations can be triggered. For this reason, even a single bit leak could cause problems, as it could be used to obtain access to secure data [80].

In a serverless environment, there are two main security drawbacks [80]. On one hand, the security level strictly depends on the features given by the vendor. On the other hand, if insecure code is used for serverless functions, the attack surface is extended. Thus, the authors accurately suggested how to choose the vendor service, paying particular attention to the quality of the code and introducing continuous monitoring of the production

environment. Researchers are suggesting approaches to improve the security mindset of the consumer, applying specific actions, such as a) introducing a culture of collaboration, b) creating security-by-design architectures [81], c) introducing threat modeling and limitation of authorization [82], d) implement secure coding standards [83], and e) automate secure deployment systems, exploiting continuous monitoring [84]. Such approaches are required in a Serverless environment to ensure comprehensive security controls based on multilevel protection [85].

In the fields of authentication and authorization, external request permissions should be verified for all the functions at the workflow entry point. This allows malicious requests to be blocked as early as possible, thanks to an authentication and authorization system based on the decoupling of authentication from execution with the use of a message-oriented middleware [86, 5], and using JSON Web Token (JWT) and Oauth2 protocol for authentication, authorization and access control in a serverless environment [87]. In [88], the authors proposed a dynamic and self-adapting approach for data access and protection during its whole lifecycle in the Cloud-to-Edge continuum. This solution was proposed to protect data in applications deployed in the continuum but not for sequential and parallel serverless workflow executions. However, a similar approach could also be adopted for this kind of application. In the field of resource isolation, the greatest challenge is represented by the weak isolation of lightweight virtualization systems, aided by virtual machines and containers to isolate functions and contexts [85, 76]. Containers' security strictly depends on the underlying operating system (i.e., a breached container instance can easily trigger an operating system vulnerability); whereas virtual machines (VMs) ensure better isolation, but they are more resource-consuming.

Nevertheless, different solutions for a lightweight and secure execution have been proposed in recent years: Amazon introduced FireCracker [89] in 2018, a new hypervisor that uses microVMs for deploying serverless functions; Google developed gVisor in 2019 [90] to completely isolate serverless function deployed in containers from the host operating system. However, even if functions are isolated by being deployed using containers, malicious code can still trigger the execution of prohibited functions. Thus, it is not sufficient to isolate the software execution, and some network isolation should be introduced. In the field of data protection, encryption is necessary to protect data both in idle and transit. Moreover, some key management services are needed to handle application secrets [85], such as cryptographic keys.

Serverless is mainly born as a natural evolution of the microservice architecture that typi-

cally runs in the Cloud, but it is common to find Edge deployment [18, 91, 92]. Unfortunately, the security risks of serverless on distributed environments are poorly treated. In [93], the authors proposed a Multi-Cloud Performance and Security (MCPS) Brokering framework for resource allocation of a set of workflows across federated Multi-Cloud infrastructures. However, even if it is possible to introduce security constraints in selecting the nodes for the computation, it is limited to Cloud infrastructure and only uses VMs. Therefore, it does not use lightweight virtualization solutions, such as containers, and does not support Edge nodes. Another proposal for a secure scheduling system of workflows was made in [94], where some security constraints were introduced for the scheduling of jobs in a four-tier environment composed of IoT sensors and devices, mist resources, fog resources, and Cloud resources. Such a solution does not have an implementation for serverless computing, but its applicability could be really important for that field.

5.1.2 Threats Analysis and Mitigation

According to the above-mentioned background in Osmotic Computing, we borrow the concepts of MELs and SDMem to implement a secure execution of serverless workflows on a Continuum environment, merging into OpenWolf components (i.e., OpenFaaS and Kubernetes). Below, we identify the main threats and discuss of to mitigate them.

Data and Communication Privacy

When OpenFaas builds and deploys a function, it is encapsulated within a container that Kubernetes includes in a pod, acting as MS. Even if Kubernetes includes the concept of secret, it is not good enough to guarantee data privacy, since this is stored in plain text, and its access is not regulated by any policy. In this regard, while implementing MOD and MUD, we need to use Hashicorp Vault, which is used to store encrypted secrets (aes-gcm 256 bit). Therefore, a client needs to get authentication and authorization according to the secret's policies before accessing the secret. This feature satisfies the Objective O1. Even if Hashicorp Vault encrypts data, this could be stolen during the transmission from the Vault to the Pod. Therefore, we use the Vault Injection to deploy a vault container in the same pod where the function container lies. In this way, data are encrypted locally. This feature satisfies the objective O2.

When OpenWolf deploys the function on Kubernetes, the Vault injection happens transparently. Therefore, the function container does not need to directly access secret data.

In Kubernetes terms, running a container that supports another one in the same pod is called the side-car pattern. In our case, we are adding a container that lets the function pod act like an Osmotic MEL, guaranteeing privacy and security to confidential data; for this reason, we call this container Osmotic Sidecar. Figure 5.1 shows how the function, the sidecar, and Vault work together.

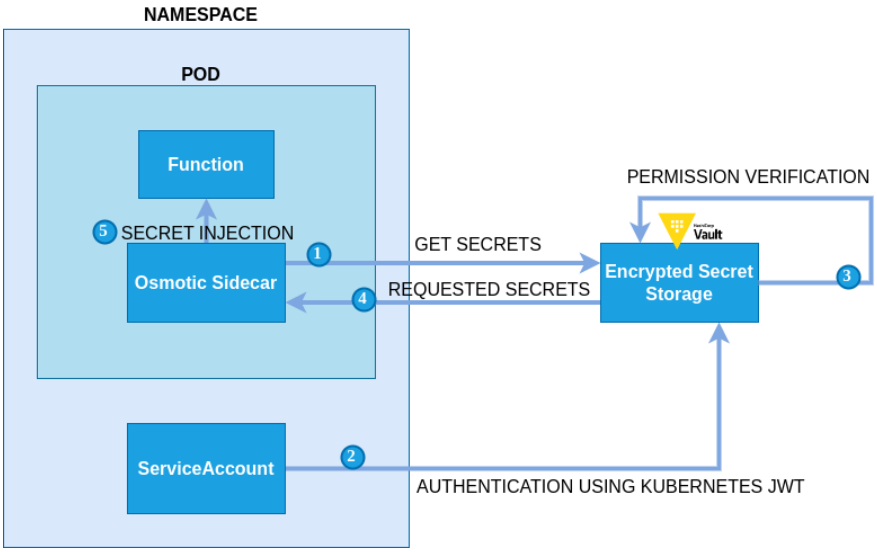


Figure 5.1: MEL design in OpenWolf

Basically, to enable the sidecar in any Function we have to follow four steps: i) enabling Vault; ii) defining a role in Vault; iii) associating a role to a function/pod; iv) granting permission to a role for accessing a secret.

Malicious FaaS Invocation

SDMem acts as Kubernetes tuning. Specifically, it allows constraining, isolating, and protecting MELs in a Continuum environment. In the OpenWolf architecture, this can be ensured by exploiting the network features of Kubernetes, which are i) the interaction with the pods' deployment phase and ii) the pod's network ingress/egress network rules.

Function interactions are ruled by the Manifest, which describes how each function sends data to others in the Workflow. Unfortunately, this approach does not prevent a malicious function from invoking other functions by hard coding its address and bypassing OpenWolf, which instead could notice an unattended connection attempt. SDMem is then designed to allow only legal interactions and reject any others. The implementation of the SDMem on OpenWolf is based on two features of Kubernetes: (i) Namespaces and (ii) Network Policies. Usually, OpenFaaS deploys any function in the same namespace, but we changed

this behavior to add an ad-hoc namespace that contains all the pods that belong to the same membrane, and these pods can interact using their relative proxy interface placed in ad-hoc separated namespaces. To do that, we denied by default any ingress policy to the OpenFaaS gateway, with the exception of the OpenWolf agent and the proxy. The agent can access the allowed functions running in another namespace that accepts ingress traffic from those functions. An example of a policy applied to the OpenFaaS gateway is shown in the listing 5.1.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: faas-gateway-policy
  namespace: openfaas
spec:
  podSelector:
    matchLabels:
      app: gateway
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: openfaas
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: openwolf
        - namespaceSelector:
            matchLabels:
              app: nginx
```

Listing 5.1: Policy applied to the OpenFaaS gateway

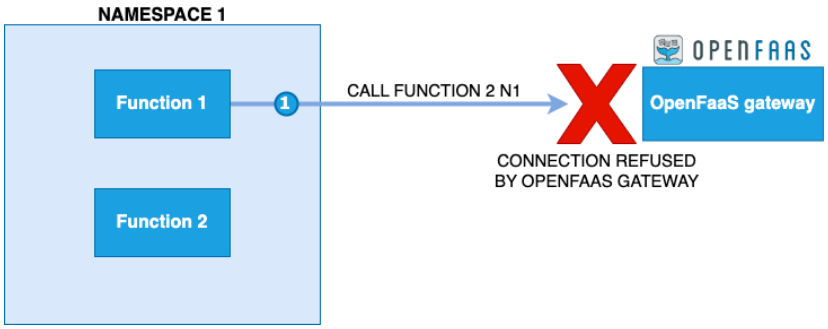
An example of a policy applied to the proxy is, instead, shown in the listing 5.2.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: proxy-group1-network-policy
  namespace: proxy-group1
spec:
  podSelector: {}
  policyTypes:
```

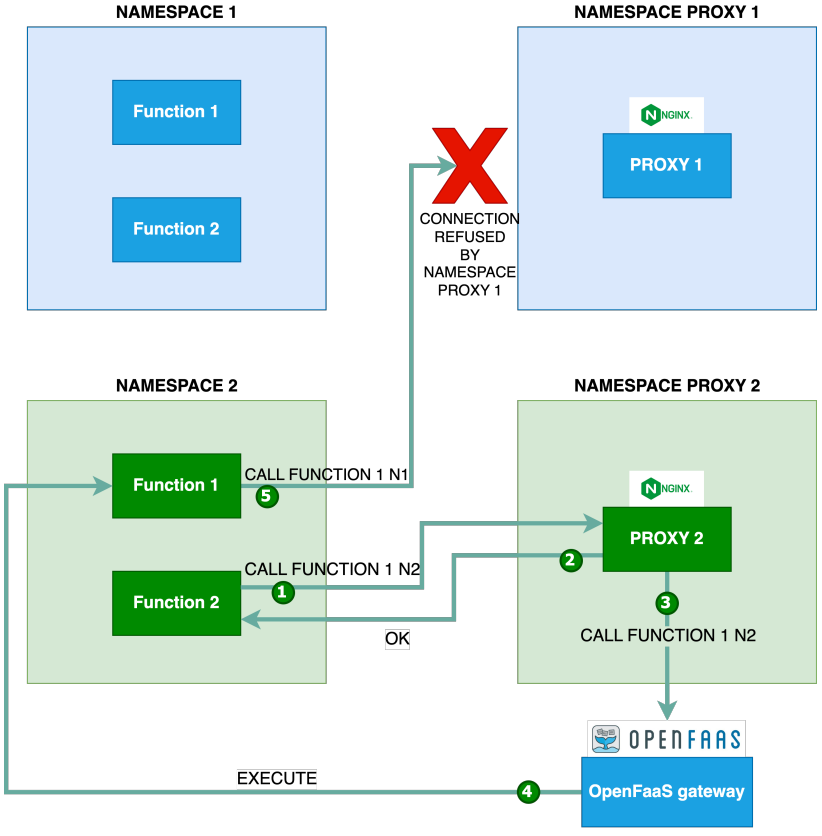
```
9     - Ingress
10  ingress:
11    - from:
12        - namespaceSelector:
13            matchLabels:
14              kubernetes.io/metadata.name: openfaas
```

Listing 5.2: Policy applied to the proxy

The behavior we shaped is given in Figure 5.2, where we graphically show how Network Policies works, basically rejecting any interaction that does not belong to the same membrane and that does not pass through the proxy.



(a) Function Gateway Isolation



(b) Proxy Namespace for MEL isolation

Figure 5.2: Function isolation

Message Security, instead, consists of the encryption of the data exchanged between functions linked in a workflow. As we already mentioned, cryptography is a transparent option for the functions which are unaware of how data arrives at the destination. On the other hand, encryption and decryption must happen in a protected environment to avoid data theft. To respect those constraints, we generate a key encryption secret for each function and then deploy this secret both in Vault within the function's Osmotic sidecar and in the agent, as shown in Figure 5.3. The Osmotic sidecar injects the key into the function container that uses it for encrypting the function's output before it reaches the pod. The Agent instead decrypts data with the same key stored in its own vault, and it encrypts this data with the key of the next function in the workflow manifest. This satisfies the Objective O5.

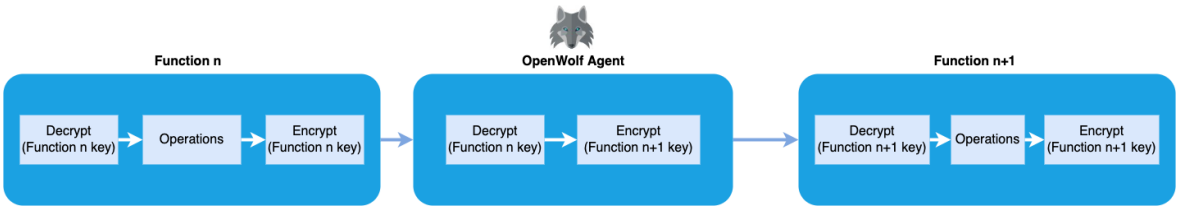


Figure 5.3: Message encryption workflow

Even if the pods' interaction is protected by message encryption, the communication is not protected. SDMem isolates and protects by design any data exchange at the network level to prevent sniffing of encrypted data that could be decrypted with a brute force attack. OpenWolf manages the Continuum Network Federation using the Kubernetes Container Network Interface (CNI), allowing the deployment of a WireGuard VPN Server and a Wireguard VPN client in each Kubernetes node, isolating the network communication from the external and then preventing stole of data, as shown in Figure 5.4. The default CNI of Kubernetes is flannel, which operates via an IPv4 overlay network. Each node in the cluster is associated with a dedicated subnet on which to internally allocate IP addresses. When a POD is started, the Docker bridging interface on each node allocates a dedicated address for each container. The PODs within a single host communicate through this bridge, while in the case of communication between PODs in different hosts, Flannel applies an "encapsulation" of the frames in UDP and performs routing to the correct destination. However, we have preferred to use a different CNI, such as Calico. Unlike Flannel, it does not stand out for its simplicity but for performance, reliability, and versatility. In fact, Calico's spectrum of action extends not only to the aspect of connectivity but also to security and network management. Calico does not use an overlay network but configures a layer 3 network using the BGP protocol for the correct routing of the packets (encapsulation is not required), with an evident

performance gain as well as facilitation in case of debugging. Moreover, Calico implements a tunnel to secure the communication channel used by nodes. In this regard, it uses Elliptic Curve Cryptography (ECC), a type of public key cryptography based on elliptic curves defined on finite fields. This also satisfies the Objective O4.

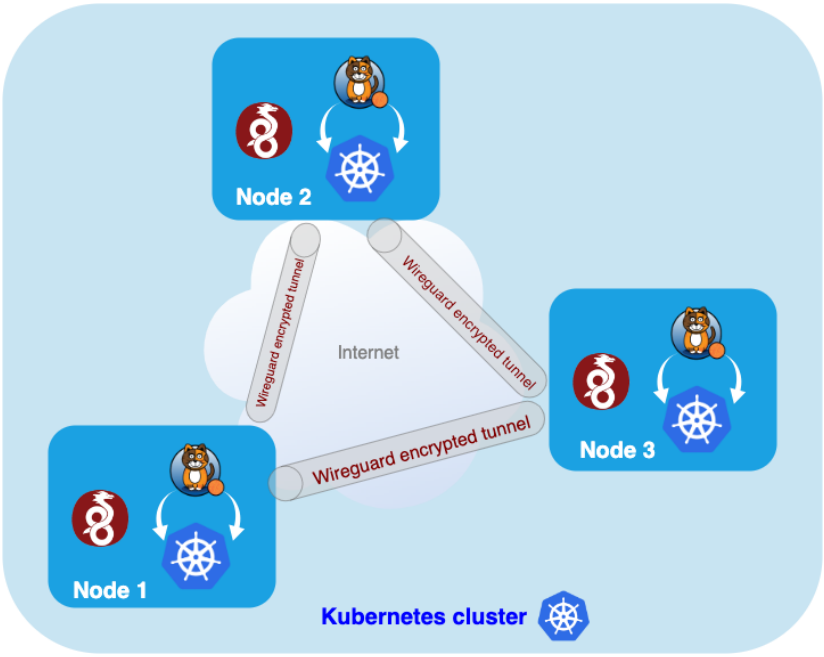


Figure 5.4: Infrastructure SDMem using VPN

5.1.3 Benchmarks

Experiments about the security-enhanced features implemented in our OpenWolf prototype are described in this Section. Compared to its original implementation, tested in a previous work [2], we expect to appreciate an overall performance deterioration (especially in the execution time) due to the encryption and decryption phases applied in both sequential and parallel workflows. However, the security benefits are worth this drawback. In the following paragraphs, different performance metrics are analyzed and compared, such as the (i) Time to Respond (TTR) also called Execution Time, and the (ii) CPU and RAM usage, considering both a variable number of states and different key encryption lengths, in a full-Cloud, full-Edge, and Continuum environment in both sequential and parallel configurations.

System Testbed

The Osmotic version of OpenWolf has been tested using a three-node Kubernetes Cluster, composed of one node in the Cloud tier, and two nodes in the Edge tier. The OpenFaaS's Gate-

way, Prometheus, Authentication Server, Kubernetes Master, OpenWolf Agent, OpenFaas' Nats, Queue Manager, and a Redis instance have been deployed in the Cloud environment, while the Edge is left empty, but available at hosting functions. Table 5.1 contains the information about our Continuum environment. Both Cloud and Edge are suitable to host all the OpenWolf and OpenFaas's components, as well as the functions that compose the workflows we used to make the test assessments. In the next sections, we compared the behavior of OpenWolf in terms of response time and resource utilization when deployed in three environments: full-Cloud, full-Edge, and continuum. In the full-Cloud test assessments, all the OpenWolf components as well as the functions that compose the workflows are deployed in the Cloud nodes, other tiers are still federated in the Kubernetes cluster, but they are idle. In the full-Edge environment, the opposite happens, with all the architecture components and functions deployed just in the Edge nodes. Finally, in the Continuum environment, the OpenWolf components are deployed in the Cloud, while the functions are fairly distributed among all the nodes.

The systems' characteristics are summarized in table 5.1, while the OpenWolf parameters are summarized in table 5.2.

Instances	Tier	Model	CPU	Memory	Operating System
1	Cloud	Openstack VM	Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, 2-core	4 GB	Ubuntu 20
2	Edge	Raspberry Pi 4	ARM64 SoC 1.5GHz, 4-core	4 GB	Raspberry OS ARM64

Table 5.1: Cluster's nodes characteristics for the Osmotic Workflow testbed

Parameter	Value	Condition
Queue Workers	1	Ever
Function replicas	State references	States Number $<$ 60
Function replicas	(State References)/2	States Number $\geq$ 60
Max_inflight	Equal to functions replicas	Ever

Table 5.2: OpenFaaS and OpenWolf parameters for the Osmotic Workflow testbed

Encryption Overhead

The first test compares the execution time of a dummy function both in Cloud and Edge with and without encryption for input and output data. The encryption algorithm adopted for evaluation of the metric is the Advanced Encryption Standard (AES), a symmetric key

block cipher algorithm. The block has a fixed size (i.e., 128 bits) and the key can be 128, 192, or 256 bits. The use of the AES algorithm is justified by the fact that it is used as a standard by the government of the United States of America and can in fact be used to protect classified information. Specifically, a 128-bit key is sufficient for the SECRET level, while 192- or 256-bit keys are recommended for the Top secret level. AES makes 10 rounds for a 128-bit key, 12 rounds for a 192-bit key, and 14 rounds for a 256-bit key. Therefore, the tests were carried out considering an increasing key length (i.e., 128, 192, and 256-bit keys).

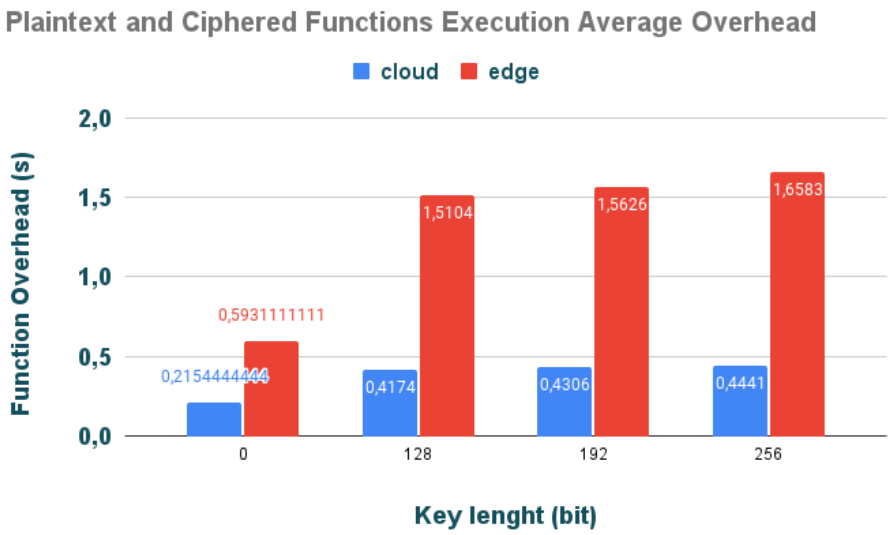


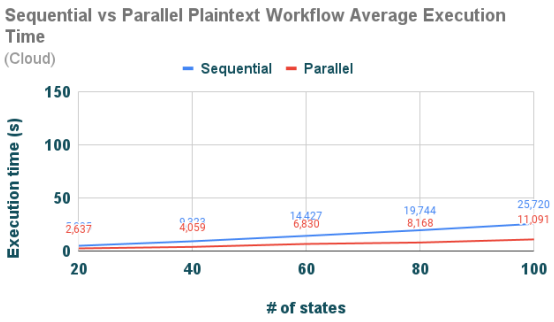
Figure 5.5: Average Serverless Function Execution Overhead

Analyzing the performance of the system when processing plaintext, as shown in Figure 5.5, we evaluate that an Edge node requires 175% more to execute the same function with respect to the Cloud node. When cryptography is applied, the Edge execution time is around 260% times the Cloud one. This constant behavior is motivated by the fact that cryptography is a near-unit cost, independent of the key dimension for a small range of keys. This is confirmed by the fact that Cloud and Edge encryption execution time takes respectively 100% and 260% more respect the plaintext execution, regardless of the key size.

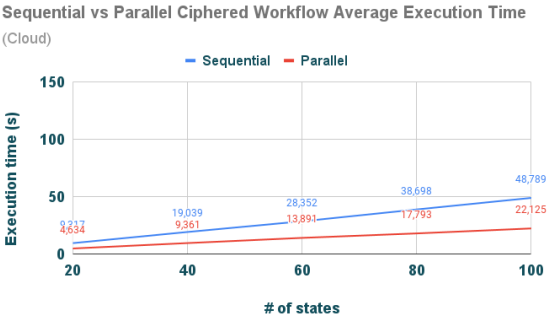
Parallel and Sequential Workflow Execution Time

OpenWolf is able to build complex functions relationships thanks to the use of the Serverless Workflow DSL. The simplest serverless combination is chaining, where each workflow function follows the previous one. With respect to OpenFaaS and OpenWhisk, OpenWolf is also able to run parallel functions at once and then join them later. This possibility can be time-saving, as already demonstrated in the previous works, but the cipher capabilities

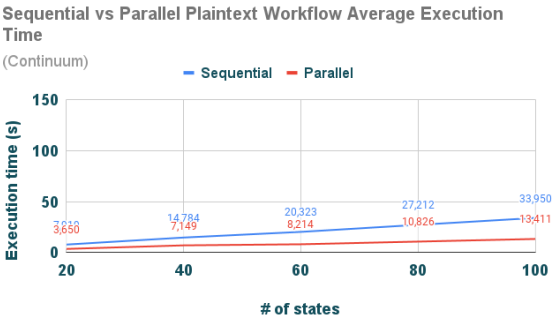
could badly affect the scaling factor from the sequential to the parallel execution due to heavier resource work. In Figure 5.6, we compare the execution time of the same workflow deployed both in sequential and parallel mode. In subfigures 5.6a, 5.6e, 5.6e, all messages are exchanged in plaintext between functions. Instead, in subfigures 5.6d, 5.6f, 5.6d, messages are encrypted using a 256-length key. Same results are even shown in the tables 5.3, 5.4 5.5, 5.6. The results obtained with this test show a good scaling factor on all three configurations. In the Cloud tier, a parallelized job can save from 48% of the time to 56% for a plaintext execution and from 50% to 54% for a ciphertext one. This result is obtained using two computing nodes. On the Edge, the improvements are similar to the Cloud. The plaintext execution saves from 54% to 64% of the time, while the ciphertext execution from 58% to 60%. Finally, thanks to the use of more computation nodes and a greater parallelization factor in the Continuum environment, a plaintext execution saves from 53% to 61%, while a ciphertext one from 62% to 68% of the time.



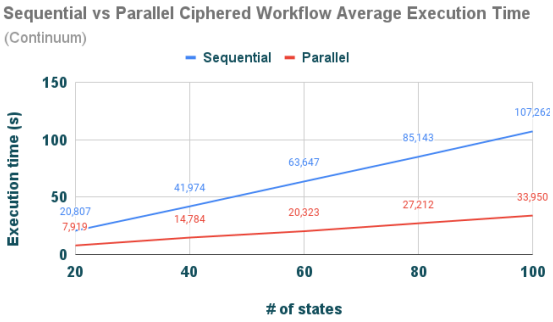
(a) Plaintext TTR Comparison in Cloud



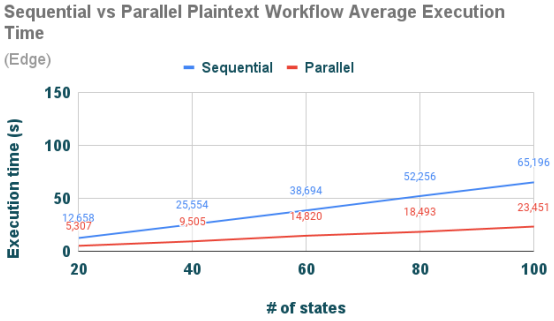
(b) Ciphertext TTR Comparison in Cloud



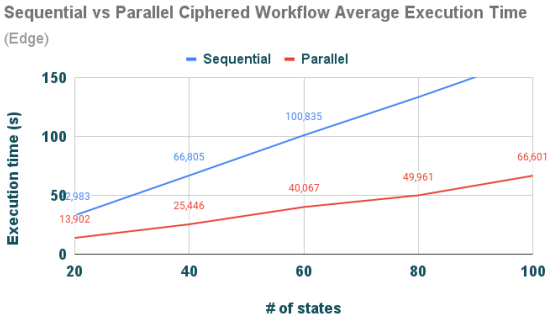
(c) Plaintext TTR Comparison in Continuum



(d) Ciphertext TTR Comparison in Continuum



(e) Plaintext TTR Comparison in Edge



(f) Ciphertext TTR Comparison in Edge

Figure 5.6: Sequential vs parallel workflows execution time comparison

Tier/States	20	40	60	80	100
Cloud	5,095	9,323	14,427	19,743	25,719
Edge	12,658	25,554	38,694	52,256	65,196
Continuum	7,919	14,784	20,323	27,212	33,950

Table 5.3: Sequential in-clear Workflow execution time summary

Resources Utilization Comparison

The main differences between Cloud and Edge are given by closeness to data, cost, and resource availability. In this test, we will focus on the latter factor to understand how CPU

Tier/States	20	40	60	80	100
Cloud	9,317	19,038	28,352	38,6984	48,789
Edge	32,983	66,805	100,835	133,193	167,219
Continuum	20,807	41,974	63,647	85,143	107,262

Table 5.4: Sequential ciphered Workflow execution time summary

Tier/States	20	40	60	80	100
Cloud	2,637	4,059	6,830	8,168	11,091
Edge	5,307	9,505	14,820	18,493	23,451
Continuum	3,650	7,149	8,214	10,826	13,411

Table 5.5: Parallel in-clear execution time summary

Tier/States	20	40	60	80	100
Cloud	4,634	9,361	13,891	17,793	22,125
Edge	13,902	25,446	40,067	49,961	66,601
Continuum	7,919	14,784	20,323	27,212	33,950

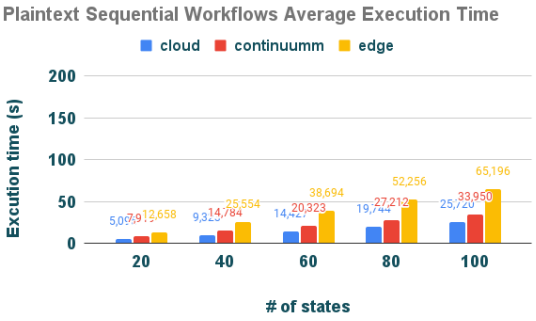
Table 5.6: Parallel ciphered execution time summary

and RAM are affected in both environments when different kinds of workflows are run on. We compared OpenWolf with the cryptography feature enabled and disabled, using workflows composed of increasing states. All states invoke the same function built to isolate OpenWolf overhead from OpenFaas's container management overhead. In Figure 5.7 we compared the metrics when a sequential and a parallel workflow are run in three different environments using plaintext data. Subfigures 5.7a and 5.7b confirm that the Cloud is better than the Edge in terms of execution time for a sequential and parallel workflow, as this performs 50% faster. The behavior at the Continuum is around the average point between Cloud and Edge, and this can be guaranteed by a good load balancing among the resources. As shown in subfigures 5.8a and 5.8b, the difference is in the gap between Cloud and Edge. In fact, the Cloud is able to run up to 4x times faster than the Edge, which delay is related to the encryption.

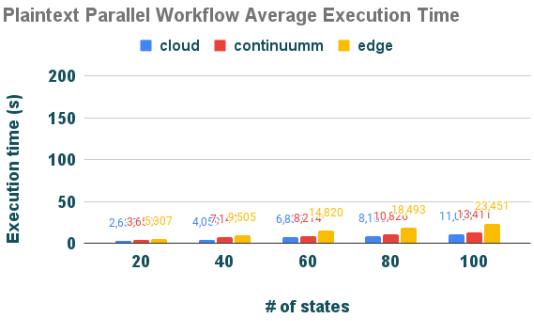
In Figures 5.7e and 5.7f we compare the CPU usage in the environments. Even in this case, Cloud is faster than Edge because of a higher clock in the CPU. On the other hand, the function parallelization involves all the cores of the Edge devices, increasing the overall usage. When encryption is taken into account, as shown in Subfigures 5.8e and 5.8f, the differences between Edge and Cloud decrease. This happens because Cloud nodes can still increase their usage, while Edge nodes' utilization is already at its max. This also explains the difference in terms of execution time as commented before. In all the scenarios studied, the Continuum environment records good performances, load balancing CPU usage proportionally with the

available nodes.

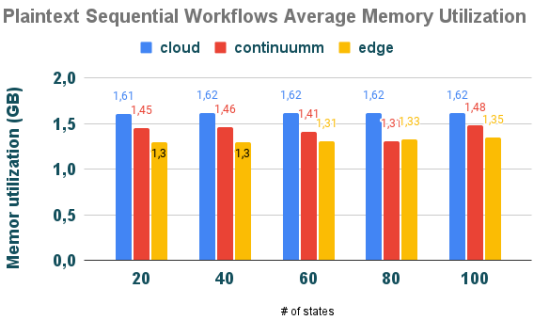
While we have seen better performances in terms of execution time and CPU utilization in the Cloud, the memory usage, as highlighted in Subfigures 5.7c and 5.7d, is in the average 26% more used. Considering that just one function is run in each node and that OpenFaas does not allow a zero-scale, this value is on average equal for any execution. Such a difference is mainly led by resource-consuming containers when compiled for amd64 architecture [95] even if idle. This is especially true when compared with a container compiled for arm64 architectures.



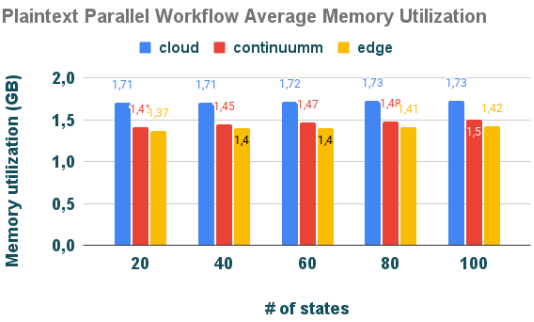
(a) TTR in-clear sequential Workflow



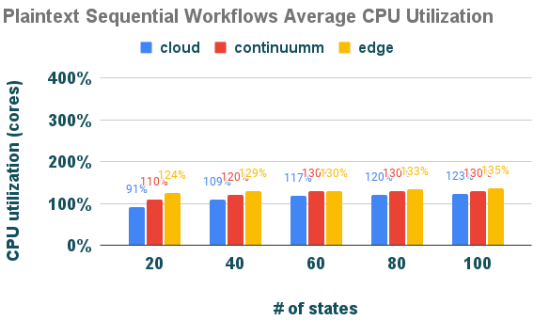
(b) TTR in-clear parallel Workflow



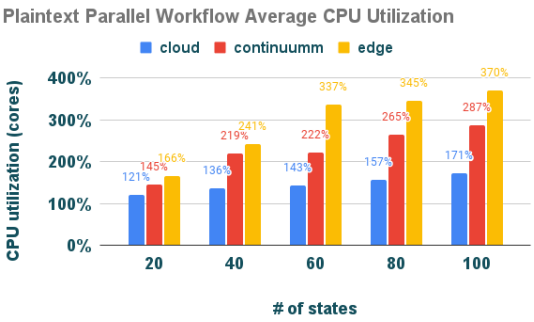
(c) Memory usage in in-clear sequential workflow



(d) Memory usage in in-clear parallel workflow

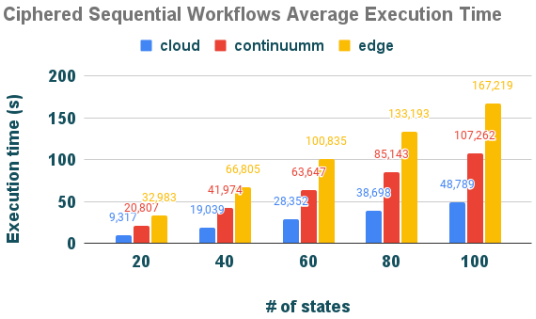


(e) CPU usage in in-clear sequential workflow

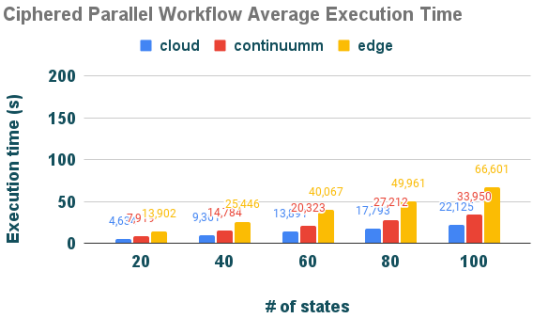


(f) CPU usage in in-clear parallel workflow

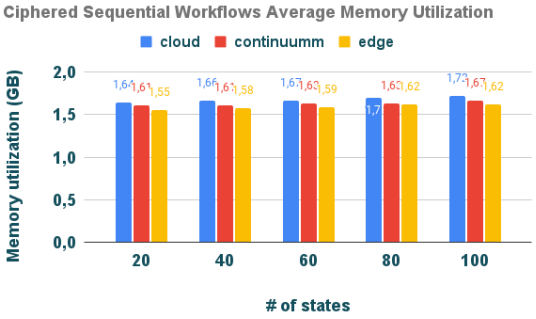
Figure 5.7: Plaintext workflow execution, sequential parallel comparison



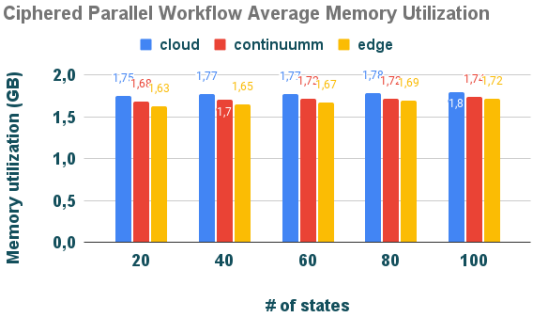
(a) TTR in ciphered sequential workflow



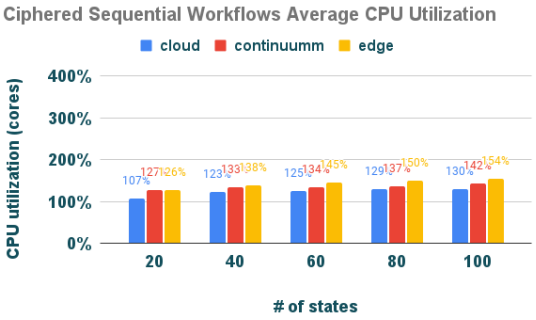
(b) TTR in ciphered parallel workflow



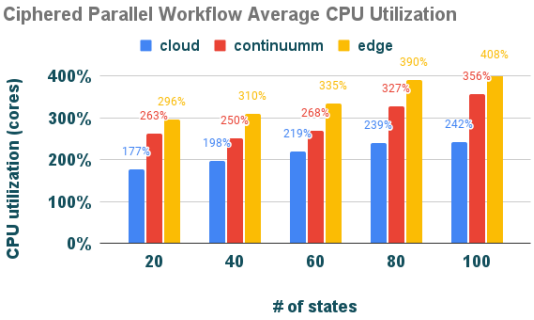
(c) Memory usage in ciphered sequential workflow



(d) Memory usage in ciphered parallel workflow



(e) CPU usage in ciphered sequential workflow



(f) CPU usage in ciphered paralel workflow

Figure 5.8: Ciphered workflow execution, sequential parallel comparison

5.1.4 Conclusions

In this work, we tried to identify and address the security risks involved in a serverless environment over the Continuum. In particular, we highlighted five different threats that are: i) the possibility to safely store secrets, ii) the ability to access them in a safe and authorized way, iii) the flow control which means the ability to allow some functions interactions and disallow others, iv) the capacity to guarantee a safe communication channel over the internet, and finally v) the capacity of guaranteeing message privacy. To address all these aspects, we followed the Osmotic Computing paradigm with the aim of providing a safe and balanced distributed environment in which running applications composed of many tasks called microservices and locating user and system data called MicroData. Osmotic Computing is safe-by-design because involves a security abstraction called Software Defined Membrane, which isolates applications and nodes. We dealt, therefore, with a real application of Osmotic Computing inside OpenWolf, a workflow engine able to spread and coordinate serverless functions deployed using OpenFaaS among the Cloud-Edge Continuum. In this regard, we deeply changed the OpenWolf architecture, modifying the underlying Kubernetes used both to build the federated network of Continuum and orchestrate the workflow functions. We performed several tests comparing the implemented features with an unsafe version of OpenWolf. In particular, we investigated metrics such as usage (i.e., cpu usage and memory) and time to satisfy different-size workflows when running both in sequence or parallel.

5.2 A Distributed Peer to Peer Identity and Cloud Edge Continuum Applications

In a continuum environment, built using the Osmotic Computing design pattern, MELs smoothly move around the Osmotic nodes following scheduling rules that can depend on performance, security, and availability aspects. These aspects have already been analyzed in previous works[6][96], but no one paid attention to the fact that if the MELs migrate, according to the Osmotic Paradigm rules, the accessing rule to the MEL might also need to be up to date. In general, any IAM demands resource access management, which defines roles and roles' access to the resources. The IAM is typically a central node with its own SQL database. This kind of architecture is hard to integrate into any Continuum/Osmotic Infrastructure, which typically is based on a distributed architecture. All the challenges we may meet using a classical IAM in an Osmotic Infrastructure are due to these architectural

differences, and they could be:

- impossibility to spread the IAM among different nodes in a peer-to-peer fashion;
- impossibility to working with some nodes offline;
- impossibility to maintain consistency over the IAMs.

The novelty of our work is represented by the presence of an IAM tailored over an Osmotic Infrastructure that tries to overcome the just mentioned limits. To do that, we will design, implement and test a distributed peer-to-peer infrastructure able to work partially offline with a soft consistency, equipped with a set of APIs that allows to dynamically change the access to any MELs according to the requests of the Osmotic Infrastructure. To do that, we will consider as Continuum baseline infrastructure, OpenWolf, the Faas-based continuum engine presented in 4.1. This chapter will mainly focus on the infrastructure level, upgrading the OpenWolf orchestrator instead of the OpenWolf engine itself.

5.2.1 Related Work

In the last years, several studies were focused on Security in microservices. The increase in applications based on this architectural paradigm was one of the main reasons to deepen the security in this field.

In the research work carried out [97] a recap of last year's studies about security in microservices architecture has been made. In particular, the need to make microservices creation and deployment secure emerges from the work. The research shows that the hotter topics about security in microservices are related to triple-A management. Even [98] solutions about security management in Microservices are treated. The paper has analyzed the security in this kind of architecture at different layers describing how each issue is treated according to the literature produced at that moment. Among the aspects faced, the solution proposed for mutual authentication by Docker Swarm and Netflix is described. The work also talks about the authorization mechanisms used in the microservices architecture. In this field, it proposes some solutions in order to manage the authentication and the authorization of final users in a microservices-based architecture. The solution proposed in the research exploits a token approach by which transmitting the authentication and the authorization among the microservices interested. A similar issue is treated in [99] which the main solutions for end-user authentication and authorization inside microservices use-cases are described. The main solutions, even in this case, foresee the utilization of a token (an example proposed is

the JSON Web Token) or, more in general, to consider a dedicated microservice for user authentication and authorization management (an SSO service). The solution proposed, even in this case, is not really innovative and considers the Identity Manager a central element. Other researchers have also discussed the authorization and authentication aspects of microservices. In particular, [100] a solution for authentication and authorization management in microservices is proposed. The architecture described in that work, exploiting the well-known defined XACML flow, aims to implement a solution in Machine-to-Machine communication in which the authorization policies are managed. The peculiarity of the solution proposed is the total microservices approach used. Each element, already known in the eXtensible Access Control Markup Language (XACML) standard, is containerized through the already validated Docker technology. The work is interesting because realizes and tests a practice solution in which microservices can manage authentication and authorization not only in a system provided for the end-user. However, it has some limits. It does not face the problem of the Identity Manager in this context. It remains a central element even in the microservices architecture proposed. Security analysis in the microservices context was deepened in the work [101]. In particular, in that paper, the security was analyzed according to different layers. On top of the layers evaluated also the application layer was studied. The solution of authentication and authorization in microservices applications was discussed, and some solutions were tested and compared. The solutions considered were a little recap of the most used techniques already discussed. They were related both to the network layer (like IP validation) and to the application one (like secret transmission). The test does not consider some important elements like an advanced authorization management system.

Another interesting work was realized [102]. In this work, a new authorization policy language is developed. It is conceptually based on the XACML standard. It simplifies that standard even using the JSON format to define the rule and the policy. This system allows the implementation of the main authorization functionalities in a microservices architecture through the "delegation" concept that characterizes the solution presented. Exploiting this new concept, the process of authorization validation can be distributed, among different microservices, without sharing sensible data, within the whole architecture making the solution really scalable and more secure. The case considered does not deep the distribution and the position of Identity Manager inside the architecture. This element should be discovered since the evolution of last year's microservices has faced.

One of the most recent papers, [103], has studied all the research in the security field within microservices-based systems. It reveals and recaps the main topic of the research considered.

They are mainly related to authorization and authentication problems as we have already seen.

The work presented in [11], described a new Computational Paradigm strongly based on the Microservices concept, that is Osmotic Computing. That paradigm tries to federate heterogeneous environments (like Cloud and Edge Computing) in order to make transparent a unique "place", to deploy services, called Membrane. This federation foresees a possible migration among the nodes federated in the membrane. This paradigm has introduced also an evolution of classical Microservices named MicroElements. MicroElements are characterized by the presence and management of Persistence Data, and they are the central element of Osmotic Architecture. Some researchers have focused on the security of the MicroElements inside the Osmotic Infrastructure. The [104, 105] basic principles of Osmotic Computing are defined and the problem of authentication and authorization is presented. In particular, the paper points out the problem related to an authentication distributed in all the federated nodes. The work proposed in our paper wants to solve this problem through the solution proposed.
In this regard, other researchers have focused their work on the decentralized management of identity and access. For example, the research carried out [106] tries to design a decentralized identity system that is based on classical credentials and that exploits Blockchain. It is a conceptually good starting point for Decentralized Identity Management, but it is not enough. It is based on classical credentials that are not many safe, and it is based on Blockchain. We can find an evolution of decentralized identity management [107], a distributed Identity Management is considered. In particular, this work takes inspiration from the Self-Sovereign Identity approach [108]. to realize a distributed identity and access manager blockchain-based in which personal data are encrypted through homomorphic encryption. The solution is interesting but it seems a little bit not convenient for the end-user. Moreover, they are not oriented in Microservices and Osmotic approach unlike the solution proposed by us.
A recap of the more important research on decentralized identity is described in [109]. The work shows all the most recent research about decentralized identity management blockchain-based. All the solutions seen within that paper are interesting but all of them are based on blockchains that could represent a third part element inside the Osmotic Infrastructure. The solution proposed in our paper is strictly Osmotic compliant and allows to manage authentication and authorization without third-party elements respecting the Osmotic Architecture itself.

5.2.2 Design

Typically IAMs, are deployed using an architecture similar to the one shown in Figure 5.9. In the figure, we represented two different nodes. The first node acts as a centralized IAM, composed of a set of access rules and user and service data stored in a database, a set of APIs for authenticating users and a proxy server used to forward the requests from the client to the correct service that instead are stored in a second node, usually located in a private network area. Typically access rules are represented like a sentence with the following shape: "⟨ Subject⟩ can ⟨action⟩ on ⟨object⟩", In these cases, the actions are typically involved in CRUD (Create Read Update Delete) set operations and the object is any available resource. The subject depends on the security pattern followed, for example, RBAC or ABAC. This topic is out of the scope of this chapter and for simplicity, we will consider starting now an RBAC-based IAM.

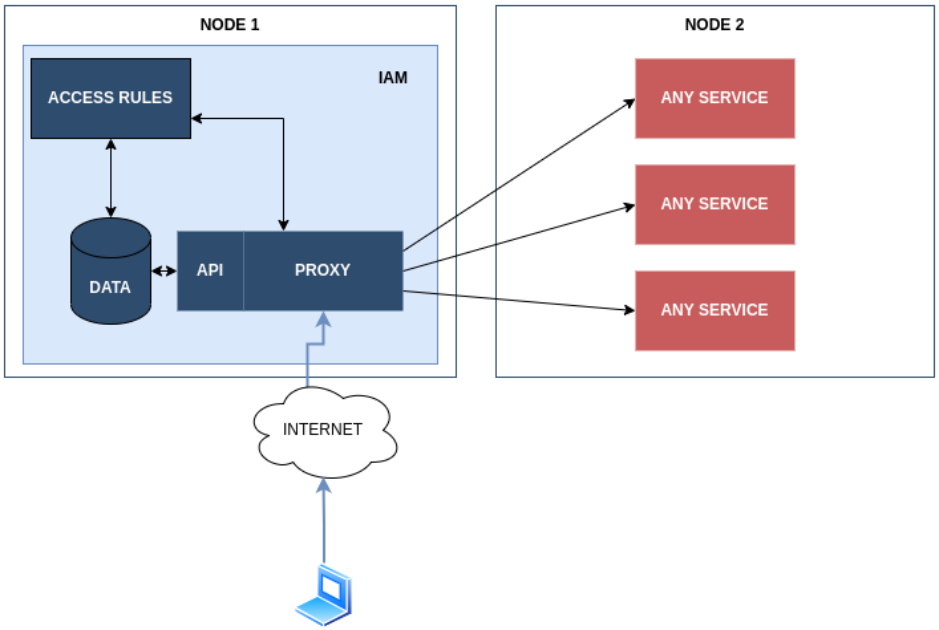


Figure 5.9: Centralized IAM architecture

In this scenario, users use the IAM's API to authenticate themself using their credentials, and the IAM's proxy to reach the services. The proxy will use the authentication information provided by the client and the access rules stored to decide whether they can reach the requested resource. This flow is well described in the sequence diagram in Figure 5.10. As we see there are no interactions between the clients and the real requested resources, all the messages in fact arrive at the Proxy and only the authorized ones are forwarded to the resource. In an Osmotic Infrastructure MELs are spread among different Osmotic Nodes as well as other Osmotic Components. Often, Nodes are not reachable between them, due to

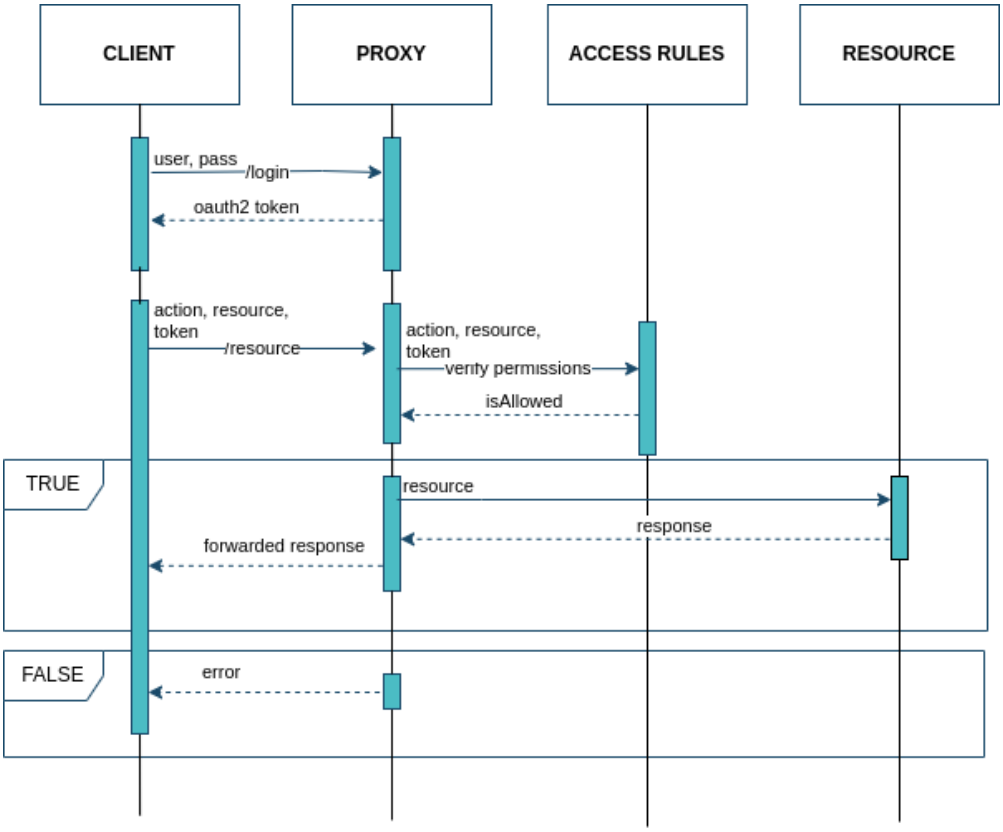


Figure 5.10: Authorization and access flow

network disconnection or security reasons related to the Osmotic Membrane [11]. Due to that, tailoring a centralized IAM in this scenario is a high risk, since it may affect the usability of some applications inside the Osmotic Infrastructure. For these reasons, we designed a fully distributed IAM as shown in Figure 5.11.

In this new scenario, we have many Osmotic Nodes, and in all of them, an IAM is installed. The IAM is still composed of the elements seen before, a database containing the user's and resource data, a set of access rules, a set of APIs, and a proxy service.

This time, the IAM is equally distributed among all the Osmotic Nodes, in a peer-to-peer model. Each node must be able to work even if any of the others are unreachable; for this reason, each IAM is in charge of guaranteeing secure access only to the local services installed in the same node where the IAM is, and eventually to the others only if they are reachable, in this way, a service becomes unreachable only if the entire node is unreachable. The different IAMs instances need anyway to be synchronized because they need to know the rules to access to service they are not hosting at this moment but they could later. To guarantee this consistency, access rules, and user and resource data are stored in a peer-to-peer database that accompanies the proxy. The consistency between the databases can be guaranteed using

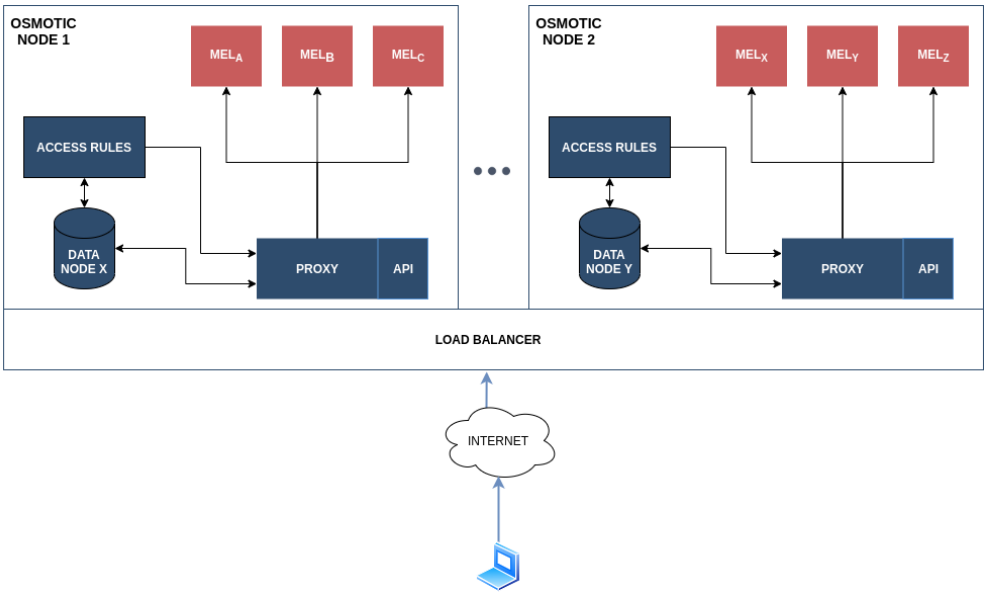


Figure 5.11: Distributed IAM architecture

a multi-version concurrency control pattern (MVCC). Finally, the data updates happen using the APIs the IAM provided. These APIs can be consumed by the MELs or by the Orchestration Engine, this depends on the implementation of the Osmotic Infrastructure, nevertheless the implementation, each API instance is authoritative only for the Node where it is installed. This behavior prevents any conflict during the data synchronization without compromising the usability of neither infrastructure and applications.

This approach requires that we know the IP addresses of the Osmotic Nodes where the services we need to contact are hosted, for usability constrain we cannot accept this. For this reason, we put in our architecture a further layer represented by a Load Balancer. This service can be deployed in replica in an SDN in order to guarantee high availability. The Load Balancer just redirects the requests coming from a client to the correct service's authoritative node, guaranteeing transparency to the final user. Finally, thanks to the use of this element we can easily reach any available Osmotic Node without caring about their real IPs. From a client-side point of view, this new fully distributed approach is transparent, in fact, the same flow described in Figure 5.10 is still valid. In the distributed approach, in fact, the client still contacts a single access point which is the Load Balancer, this element will forward the request to the correct resource authorities proxy, which will manage the authentication and resource's access flow.

5.2.3 Implementation

Following the designed architecture shown in Figure 5.11, we are going to implement it, in order to obtain a full working prototype.

The infrastructure we rely on is the same OpenWolf; in particular, we exploit the K3s Kubernetes infrastructure to deploy and integrate our IAM. Thanks to this engine we can easily deploy MELs using Pods, and we can orchestrate Pods using the Kubernetes APIs. We managed the Osmotic IAM as a multi-container pod instanced like a daemon, namely a Pod deployed in every node of the cluster.

We implemented the IAM using a distributed peer-to-peer database and a full stateless process able to interact at any time with the local database instance. This process is in charge of exposing API to interact with the security roles stored in the database and to proxy the requests towards the requests MELs (once their access is permitted).

Database

Finding a database able to meet our requirements is not easy, since most common and used SQL and NoSQL databases horizontally scale only using a master-slave model, like MySQL and MongoDB for example, and often this means that all the writing operations pass through the master and all the reading operations might need to be approved by the Master. This approach does not fit our model since we need to write and read to/from any node at any time. A good choice for our requirements is represented by CouchDB by Apache. CouchDB is a document-oriented database, that can be deployed inside a Cluster using a peer-to-peer approach. Each peer can contain a part of or an entire dataset, but it does not require keeping a stable connection with the other peers continuously, data in fact are considered eventually consistent, because peers will synchronize them as soon as possible, without disallowing updating during network disconnection periods. Once the peers are reachable again, they will use a consensus protocol that exploits the MVCC data model to keep the last updated documents.

Finally, we are going to use CouchDB to store the accessing rules the resource, and the user information.

API Set and Proxy

API and Proxy are both web services that need to interact with the same data components and with the same transient data, for this reason in the implementation phase, we merged the

API server and the Proxy into a single service that acts all the IAM operations we described until now. This service needs first of all to authenticate and authorize users, and therefore to update users' data information inside the CouchDB instance hosted in the same node. The service authenticates using a Role-Based Access Control and Cookies to maintain a connection state. The APIs involved in this are identified as Authorization API.

Authorization APIs are used to guarantee secure access to the Security API. This new set of APIs is in charge of updating the Access Rules which the node is authoritative.

Finally, Authorization and Security API are used by the Proxy for processing a request coming from the internet. First of all, the Proxy fetches the cookie provided in the request and sends it to the Authorization API which will reply by providing the role of the user who owns the cookie, this role will be forwarded to the Security API with the address of the requested resource, next the reply will tell the Proxy if the request is authorized or not. Finally, the proxy will forward the request to the resource or will reject it, sending it to the client in the first case the service's response, or an error response in the second case. We implemented this component by exploiting NodeJS, Express, Nano and SuperLogin packages node.

1. ExpressJS is a very famous framework used to manage an HTTP server using NodeJS; it is used to authenticate and authorize users, expose API for updating the accessing rules, and finally for proxying the requests to the MELs.

2. Nano is the official CouchDB's Javascript library that allows database interaction;

3. SuperLogin is a quite popular microframework for Identity Management based on CouchDB and NodeJS; it works also like Express middleware, enabling the authentication and authorization policies (customized by us in order to interact with the dynamic rules).

Deployment

The distributed IAM needs of course to be integrated with the Osmotic Infrastructure realised using Kubernetes. The proxy and the CouchDB node instance are strictly dependent and they cannot work if the other one is stopped. For this reason, we used a multi-container Pod to deploy the IAM, composed of the CouchDB instance, the Proxy instance and a single run configurator that ensured that the proxy and database are well configured.

As we mentioned before, we need to deploy a single IAM instance in any Osmotic Node, To do that Kubernetes provides a sort of Deployment method called DaemonSet. "Like other

workload objects, a DaemonSet manages replicated Pod groups. However, DaemonSets attempts to join a one-pod-per-node model across the entire cluster or a subset of nodes. When you add nodes to a node pool, DaemonSets automatically adds Pods to the new nodes as needed"[110].

Load Balancer

The Load Balancer is strictly related to the Osmotic Infrastructure implementation, in this case, Kubernetes provides the Kube-Proxy which is installed on each Kubernetes cluster's node and acts as a proxy for UDP, TCP and SCTP communications. It is used as the entry point for reaching the Osmotic Services, for this reason, any other further design choice like the use of a service's authoritative IAM is demanded from the IAM itself instead of from the Load Balancer.

The final result is shown in Figure 5.12.

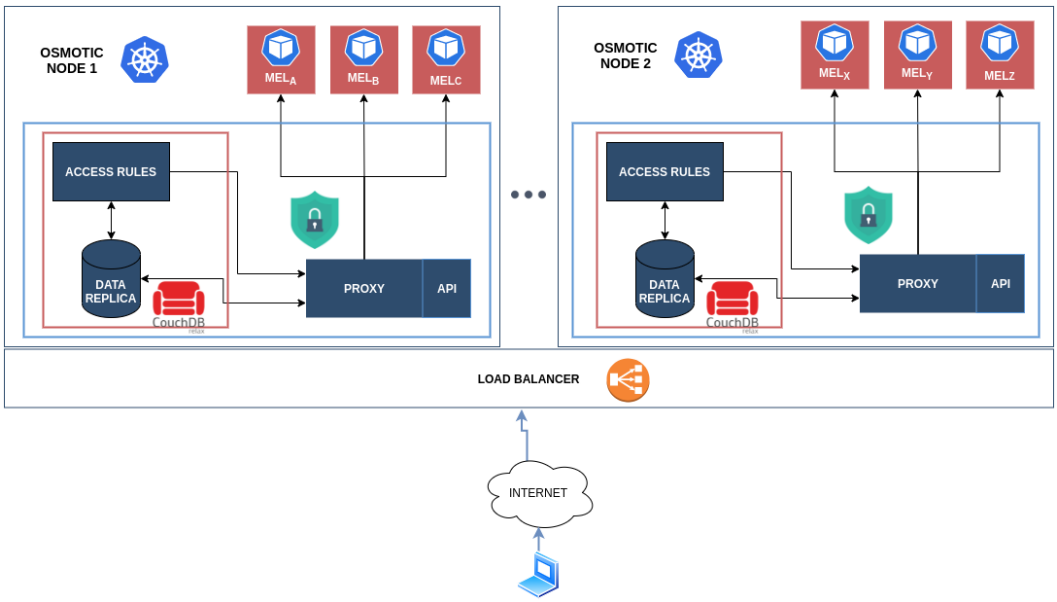


Figure 5.12: Osmotic IAM architecture

5.2.4 Use Cases

In this Chapter, we are proposing a solution for guaranteeing secure access to any application deployed inside an Osmotic Infrastructure. To keep secure access to these applications, we designed a Proxy that, through a series of rules, which are shared among all the nodes and stored in a local database that, in an RBAC fashion, allows or denies access to the resources.

The first concern about Osmotic Computing regards the policy dynamic; in fact, in

an Osmotic Infrastructure, any access rule might fast change due to service or network constraints; for this reason, we provided our Proxy with a set of APIs that allows adapting the proxy to the new rules.

The second constraint in Osmotic Computing is network disconnection. In a fully distributed environment, sometimes some nodes could be isolated from the other ones, but this should not limit the use of the node itself. Since nodes share policies and data, they should also be able to synchronize them as soon as they can interact again each other. For overcoming this limit, our IAM has been also designed in an Osmotic fashion, decoupling it into as many replicas as the nodes are and using a peer-to-peer database able to maintain an eventual consistency and to synchronize it with the other database peers as soon as possible, without denying the usability.

In this section, we will present two simple use cases where Osmotic Computing is applied where the just-mentioned problems might present and how our solution solves them.

5.2.5 Smart City Use Case

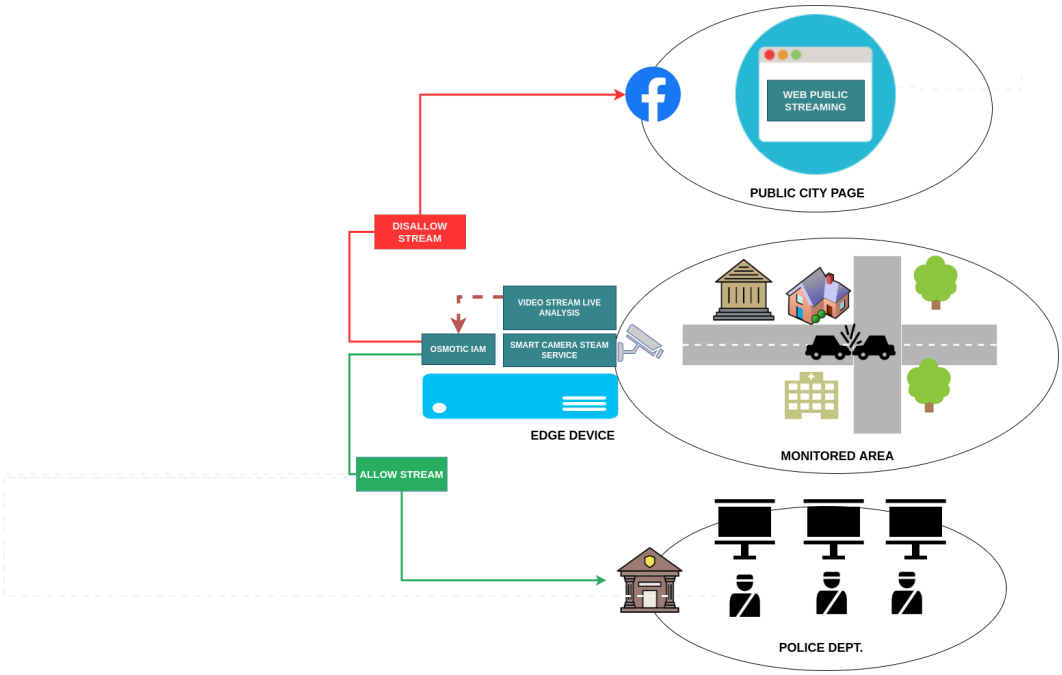


Figure 5.13: Smart City scenario for the Osmotic IAM

Nowadays, Smart Cities are characterized by many common features, like distributed access points, environment monitoring, tourism, and security tech services. For example, smart video stream services are used to show some important city areas, museums, or parks directly from the smartphone or video panel spread around the city. Cameras and video

streams are also used to maintain some areas under control, and usually, these cameras are used by the Municipality's policy or any other security department [111]. Of course, in the first case, access to the camera can be guaranteed to any public with or without authorization. Instead, in the latter case, only the public, with proper authorization, should access the video stream. These policies are quite static, but what happens if, under the eye of a public camera, a crime happens, showing on a grand scale sensitive content? Unfortunately, this could be the case of some terrorist attacks that happened last year in Europe, where the images traveled around the World and have been shared on social media like Youtube and Facebook. As proof of concept, consider Figure 5.13; we are describing an Osmotic node installed near a city area with a cross street. This osmotic node runs a smart camera that records the scenes and a video stream analyzer that recognizes the recorded events. The streaming video is shown on the city's public Facebook page. Unfortunately, an incident happens; hence the video stream analyzer recognizes this event and immediately updates the Osmotic Proxy installed in the same Edge node, restricting the video stream access only to Public Security City's officers and disabling the video stream on the city webpage. The use of an Osmotic IAM could have solved this kind of inconvenience; in fact, as soon as a suspicious event would have been detected in the video stream, automatically, the access to the video stream resource would have been updated, and the camera would have been used only by a strict number of people involved in the city's security avoiding any dispersion of sensitive content.

5.2.6 Rural Area Use Case

A rural area or countryside is a geographic area that is located outside towns and cities, where typically the provided services are very poor, and the connectivity is quite limited. An interesting research topic that has been born in the last years regards the possibility of enriching these areas with cheap Edge-based services able to provide first-need raw services for healthcare, industrial automation, and document sharing. In this kind of space, typically deployed meta-industrial services, or first-need services, like document sharing with other areas in the same environmental conditions.

This work finds a concrete case in this situation. As we already said in our work, Identity and Access Management is traditionally centralized. The approach proposed in our work goes beyond this concept. By decentralizing these operations, our solution can represent a valid alternative to the reality we are considering. A hypothetical private resource can be protected in a Rural Area even if the Network service is not always efficient. In the typical situation in which Authentication and Authorization are centralized, the Network service

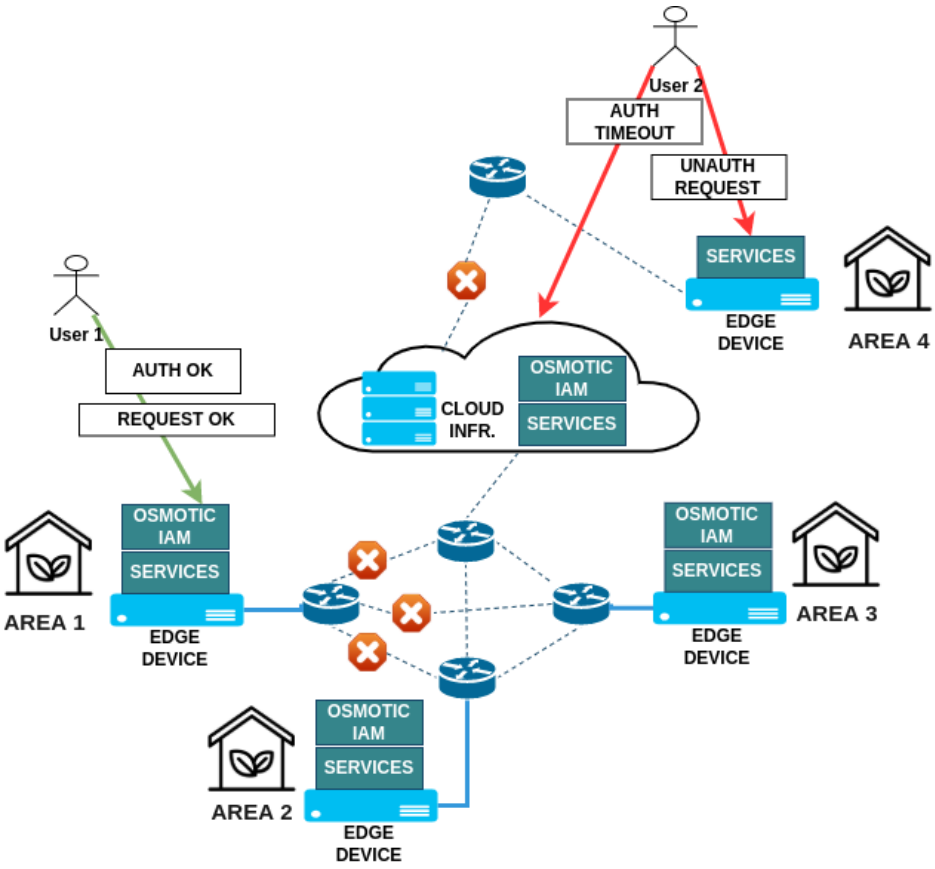


Figure 5.14: Rural Area scenario for the Osmotic IAM

absence could affect the resources' private access. Indeed, the typical systems, in this situation, do not allow resource access because they cannot verify the correct roles of the users, and, in general, authorization operations cannot be performed. In a real use case in which our solution is applied, the Authorization could not be conditioned by the global reachability of the local nodes. For example, we can imagine different systems IoT-based like a typical Smart Agriculture context that descends private data to the Edge Layer. The data detected could not be available for all entities that can physically access the Edge because, for example, they could be related to a particular Business advantage. Figure 5.14 shows the concrete context. In Area 1, for example, the user wants to access a particular resource reachable in his area only through a local connection because Internet access is not available. In typical situations, the Identity and Access Management are not able to determine the authorization attributes or the roles of the user, and it is not able to understand if the user can or not access the resource because, normally, the Identity and Access Manager are centralized and placed in a remote server only reachable via the Internet. As shown in the Figure, the architecture proposed in this work solves this issue.

The Osmotic IAM solution we have described in this Chapter could be applied on the Edge

Layer, relegating to the latter Authentication and Authorization operations. Even if the Local Edge devices are not reachable worldwide, they can authenticate and authorize the permitted users.

based

5.2.7 Conclusion

Osmotic Computing is a very interesting research topic that aims to integrate cloud, edge, fog and IoT in a unique Computation environment where applications are free to move following different concentration rules based on many internal and external parameters, i.e. Continuum. This goal is really ambitious because can enable many different use cases related to Security, DevOps and Smart Cities, but due to its complexity, many limits are still not solved at all. Two of these limits regard the access policies for the application and users and the resilience to network disconnection, the Osmotic Computing paradigm, in fact, foresees that an application can evolve, changing the node that hosts it, the active configurations (Micro Data) and the consumers. From an architectural point of view, Osmotic aims to distribute the applications among different nodes with equal roles, but different characteristics. This fully peer-to-peer design could fall into partial network disconnection, where only a few nodes can communicate with each other and in many cases, this represents a reason for some trouble. In the previous chapters, we already demonstrated how to distribute and orchestrate the computation among the COntinuum using Faas Workflow protected by the Osmotic Computing paradigm, but we have never dealt with the dynamicity of the security aspects.

In this Chapter, we presented a new Osmotic Computing component that acts like a Distributed Identity and Access Management. The IAM is itself a peer-to-peer component thought to be run in every node of an Osmotic Infrastructure. It is designed to be resistant to any network disconnection, in fact, each IAM instance works independently from the other ones but synchronizes with others' access rules and data as soon as the network allows, therefore it keeps working at any time and guarantees authorized access to the Osmotic Application at any time. To guarantee dynamic access policies for accessing the Osmotic Applications, we enriched the IAM with a set of APIs authoritative for the node where they are hosted, these APIs allow updating any rule inside the IAM and they can be consumed by the Osmotic Engine, by the application or by external but trusted clients. Immediately, once a rule is updated the IAM reflects it in the Proxy used to access the resources, and in parallel, these rules are sent to the other node to keep synchronized over the entire infrastructure.

CHAPTER 6

Discovering and Addressing Applications in a Continuum Infrastructure

We know the Osmotic Computing paradigm that gives guidelines to deploy applications able to auto-balance themself and smoothly move among all the available infrastructure in a Continuum environment. We adopted these principles in our works, and as we see in Chapter 4, we can deploy these kinds of applications using serverless workflows, but we cannot locate and relocate them when they migrate. In this chapter, we address this issue, proposing the Osmotic Computing Enabled Domain Name System (OCE-DNS), a continuum native DNS able to reference MELs using the Extended Plus Codes, a three-dimensional geocode algorithm defined by us. Experiments show that OCE-DNS guarantees quick Resource Records (RR) readings and updates, thence supporting the management of transparent Osmotic MEL migrations in a Continuum environment.

6.1 Introduction

In section 2.5, we introduced Osmotic Computing, a framework able to smoothly balance the computation at the continuum. Some use case of this paradigm has been treated in [112], [113], and usually, many of them involve smart city environments. A typical scenario is the smart video surveillance controller installed in a city square. It could comprise web services, sensors, and smart cameras with video streamers, analyzers, and collectors. These services should allow for keeping the square under control, alerting if something happens, such as peaks in audio sensor data, dangerous images recognized in the camera stream, or if a citizen

reports something bad using the local web service.

Typically, in the beginning, the aforementioned services can be entirely run in an Edge environment, but what if the surrounding environment abruptly changes its state? We could have computation congestion in the video analyzer, the web server may no longer be able to reply to all the requests; thus in brief, the square would no longer be under control. This scenario should not happen in a continuum environment. In fact, the smart video surveillance controller can be easily decoupled in a series of MELs interconnected over an isolated network. As soon as a MEL goes under pressure or attack, it could be migrated into a Cloud node where the computational resources are enough. In the end, all the MELs would return to Edge when the environment returns to a quieter state. The issues on that are mostly two:

1. How can I discover services that are geographically bounded to a specific edge area if they are not run on it?
2. How can I keep track of a service if it migrates?
3. How can the continuity of service guaranteed if a service migrates?

This chapter investigates how a decentralized naming system with advanced geocoding capabilities can support osmotic orchestrator activities. Specifically, a geocode is a unique code that represents a geographic entity that distinguishes it from others. A geocode-based naming system for Osmotic Computing should satisfy three important requirements:

1. naming in human-readable form both MELs and whole Contiuum-based applications using Uniform Resource Identifiers (URI) according to the geographical area, domain, or workspace where they run;
2. discovering MELs and/or Continuum application's services through the specification of a particular geographical workspace;
3. reacting in real-time to MELs and application migration or relocation, properly updating the naming reference.

Many works exist in the literature for naming geographical areas in a human-readable fashion [114]. However, some of them create conflicts in geocodes [115], others do not allow the definition of different size areas [116], and others are proprietary solutions [115] thus they cannot be widely adopted or integrated into open source projects. The main contribution of this Chapter is presenting Osmotic Computing Enabled-Domain Name System (OCE-DNS), which is an innovative naming system that implements a distributed Resource Records (RR)

structure for storing geocodes related to MELs and whole Continuum applications. Also, we introduce an OCE-DNS geocoding algorithm able to encode a geographical area that will be associated with MELs or whole Continuum applications, thus identifying geographical areas, domains, or workspaces even of different sizes.

Naming a MEL allows us to identify and reach it over the Internet. However, MELs (as explained above) could migrate across the continuum within an SDMem. Migration means that MELs change their physical locations, but without changing the workspace they are serving. Thus, OCE-DNS implements a geolocalization aware naming to track MEL deployment changes to guarantee performing Continuum native applications. This task is a very challenging feature that avoids inconsistencies between the pointers of the DNS and the Osmotic node where the MEL is physically migrated.

Our solution makes use of geographical and topological information associated with each MEL to build a logical representation of MELs acting within an SDMem. This allows us to reach each MEL through the well-known hierarchical DNS approach and, at the same time, according to the geographic information associated with it [117].

Using a dynamic RR database, the OCE-DNS allows us to:

1. support MEL migration by properly updating the DNS entries and guaranteeing consistency between the position of the Osmotic node running the MEL and the domain name that points to the associated Osmotic service or application;

2. associate an Osmotic service or application to a geographical workspace by logically mounting a Cloud service on the Edge and vice versa, thus enabling the *virtual device* concept (i.e., virtual Edge devices actually running over the Cloud).

6.2 State of the Art

Several works proposed the use of Osmotic Computing for smart cities. An urban pollution monitoring system based on Apollon, a piece of Osmotic Computing middleware that integrates IoT sensors with Edge and Cloud resources has been discussed [118]. An Osmotic city traffic system that allows autonomous vehicles to self-establish the priority at intersections by exchanging information with a smart traffic management infrastructure has been proposed [119]. Osmotic smart parking has been proposed in [120], where the authors analyzed the applicability of Osmotic Computing for smart parking considering different workloads. A piece of Osmotic Blockchain-based framework for smart cities has

been proposed [121]. It is composed of four layers: physical, Blockchain, Cloud, and Edge. The Blockchain layer acts as a membrane that verifies the data integrity during the migration from the Edge to the Cloud and vice-versa. All aforementioned works show possible use cases for Osmotic Computing, discussing challenges and solutions from a theoretical point of view. This work proposes a software component that supports managing Osmotic application where MELs are migrated.

Privacy and security concerns of Osmotic Computing and possible solutions have been discussed in [122], where the authors proposed the use of Secret Share techniques for storing the images of MELs on Osmotic nodes without using central repositories. The work is complementary to ours because the authors are not referring to running nodes but to their images.

The authors employed Osmotic Computing as a tool in all the aforementioned works. In this work, we discuss the OCE-DNS analyzing the time required for the propagation of the information related to the resource migration.

In recent years, with the advent of Continuum Computing, more and more researchers investigated how to minimize the delay in the propagation of information related to resource migration. A comparison of four authoritative DNS systems has been discussed [123]. The authors investigated the usage of resources and the response rate considering increasing workloads. An investigation of Managed DNS (MDNS) has been discussed [124]. Considering 8 different MDNS, the authors analyzed the delay in the propagation of the information due to a resource migration. The experiment proved that the propagation delay usually is 10 seconds. We think that the time required for updating current DNS systems are too high considering Osmotic Computing infrastructures where, depending on the use case, the MELs could migrate from one node to another one frequently, even more than once every second. This chapter demonstrates that OCE-DNS can track any migration within 100 milliseconds, even considering very dynamic environments.

Challenges and opportunities of DNS systems for IoT devices have been discussed [125]. Nowadays, one of the main challenges of these systems is related to resource transparency because the users are unaware if the interaction among local devices takes place by using remote services that could inject malware or redirect the communication to malicious hosts. In our proposed work, the OCE-DNS is a component running inside the Osmotic membrane. Therefore the security of the whole architecture is assured by it.

An inter-domain routing system for Mobile Ad Hoc Networks is discussed [126]. In particular, the author proposed a communication protocol based on resource clustering and

packet forwarding. Each resource, for the neighbor, will interact with a special node that acts as local DNS. Despite the work being very useful for advancing the State of the Art because it provides dynamicity and scalability to the system, the proposed idea cannot be employed for Osmotic resources because the system works at the routing and not at the service level. Instead, our work aims to use the standard internet network but use a host naming based on the application the host is offering and on the area in which this application works. The interaction between the nodes, therefore, takes place not only because the node exists in that geographical area but because it offers the application we are looking for in that area.

A piece of middleware for parallel container migration has been proposed [127]. The authors' framework, by analyzing the available bandwidth, estimate the most suitable number of containers that can be migrated in parallel to optimize both the stand-alone migration time and the total migration time. Despite the work theoretically being suitable with Osmotic Computing because the platform can interact with any container virtualization tool, in practice, this is not true because the time required for the resource migration is too high. That is, on average, a single container needs more than 20 seconds to migrate. As we will discuss in Section 6.6, OCE-DNS can track any migration within 100 milliseconds even considering very dynamic environments.

A similar work for VMs has been proposed [128]. The authors investigated techniques for reducing resource contention, they discussed the impact of workload characteristics on migration time. The experiments show that the migration of a single VM Even, in this case, the time required is too high considering the dynamism of Osmotic Computing.

6.3 Motivation

In the last years, with the advent of Big-Data, data mining techniques are becoming increasingly useful for discovering insights into data and enhancing user experience in services exploitation. Usually, data mining algorithms require a huge amount of computational resources and, therefore, are run on the Cloud. The introduction of Edge computing devices moved the computation power closer to end-users, thus reducing end-to-end delays in data processing. However, Edge devices' computational capabilities are still not comparable with Cloud's ones. Therefore, depending on the amount of data, some tasks have to be carried out on the Cloud anyway. Continuum is born to provide continuity between Cloud and the Edge, but the implementation of a Continuum environment is up to the developers and researchers. Osmotic Computing is able to guarantee the Continuum of moving computation from the

Cloud to the Edge and vice versa, according to the requirements of applications and available infrastructure. This is a very useful opportunity to support data mining requirements, optimize computation activities, and reduce delays in computation.

MELs associated with a specific Osmotic application flow into a single logic ecosystem called SDMem according to the rule defined by the orchestrator. These rules depend on many factors from inside and outside the SDMem as security, network partition, load balancing, process needs, etc. All these factors have an impact on the performance and functioning of the MELs themselves.

One of the most important factors for the management of continuum services is the geographic workspace. It identifies the geographical location it operates, for example collecting data from the surrounding environment, producing added-value information, and offering services that make sense only in that particular location. Difficulties in knowing the geographical context of a MEL workspace limit a lot of the opportunities brought by the orchestrator. Thus, orchestrator migration rules cannot take into consideration the geographical area in which a MEL works and the relative impact at the application layer. For example, a migration should or should not take place based on how close the node running the MEL is to the specific context. Another example could be the opportunity to move a MEL closer to others working in the same context.

A second issue related to the Continuum is naming services (MEL in the Osmotic dictionary), that is the capacity to identify MELs inside a specific SDMem. This issue affects the MEL migration process among different nodes in the same SDMem. Going in deep, an Osmotic Infrastructure can enable the migration of MELs by exploiting different approaches. MELs can be implemented through well-known software virtualization systems such as Containers. Available solutions to smoothly migrate MELs among osmotic nodes are orchestrators such as Kubernetes. Less investigated approaches are based on the use of Overlay Networks for the federation of SDMems. This last approach would lend itself even better to the Osmotic paradigm as it is strongly decentralized and independent, not based on the capabilities of a single orchestrator. Regardless of the approach used to enable an Osmotic infrastructure, we need to ensure the transparency of the MELs migration. A MEL migration process can be defined as transparent when the migration is not perceived by external clients who are communicating with the MEL itself. It derives from the implementation of hot migration at the infrastructure level, and the terms aim to emphasize that the user's quality of experience doesn't suffer from MEL migrations. The most advanced orchestrators can guarantee this transparency thanks to an intelligent request routing algorithm, but the naming systems

are not, however, based on the geographical context of the MELs themselves, which instead becomes crucial for the identification of the service and its context.

Transparency is necessary to ensure the continuity of communication with a MEL even after its migration. Let's consider the example in Figure 6.1. We consider three MELs running

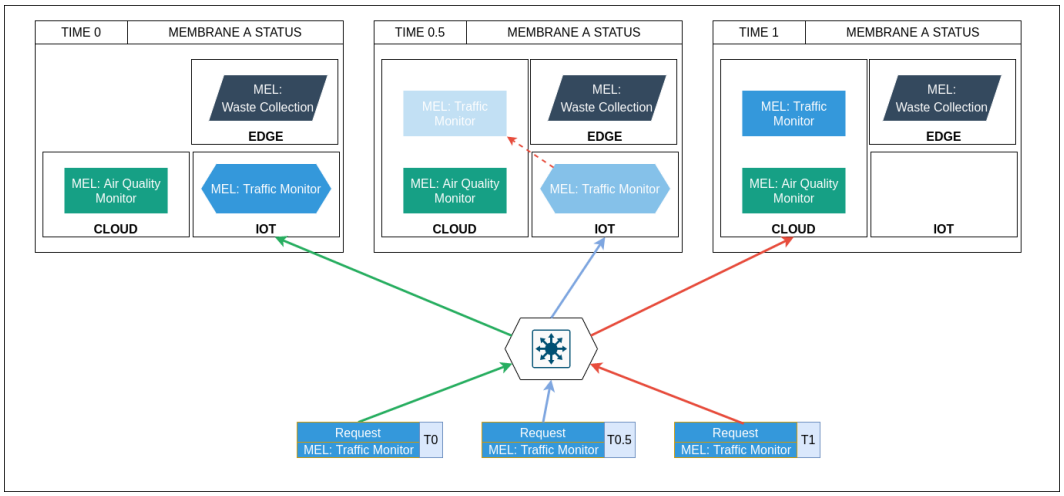


Figure 6.1: Transparent migration of a MEL

into IoT, Edge, and Cloud nodes and providing different services. For example, at Time 0, the IoT node is running a Traffic Monitor MEL. At the same time, an access request to this MEL arrives in the system and is routed to the IoT node. At Time 0.5, the Traffic Monitor MEL is migrated from the IoT node to the Cloud one, but this process needs time, and, thus, access requests coming at that time are still routed to the IoT node that does not destroy the MEL yet. At Time 1, the Traffic Monitor MEL is completely migrated from the IoT node to the Cloud one. So, whenever a new access request arrives for the Traffic Monitor MEL, it is automatically routed to the Cloud node. With this approach, the migration is transparent since the client has sent a request for the same MEL, but the request has been automatically routed to a different node.

This work, therefore, aims at providing geographical information about the context of a MEL. This information should be used by the orchestrator and by any third client; in particular, the orchestrator could define the migration rules according to the constraints that a MEL monitor already provides, and the third client could use this information to fetch, discover and interact with the MEL serving in the area the client is interested in. The geographical context information should be provided as the name of the MEL itself to allow identification of the MEL workspace and the MEL itself. The naming system should abstract the real position of MEL inside an SDMem, to guarantee the MELs' transparent migrations. We can reformulate these problems in two points:

1. Identification of a geographical context through a language understandable to humans.
2. A naming service for the MELs that can be derived by the workspace.

To do this, firstly, we intend to define a geocoding protocol that allows for identifying a workspace context. This geocoding algorithm must have a hierarchical and variable precision structure. The first feature will be useful for identifying prefixes in the code that remain constant for a given area, making the memorization process easier. The second feature should allow us to identify more or less large areas to identify size-variable workspaces.

To achieve this goal, we will use the Open Location Code, a geocoding algorithm capable of generating codes called Plus Codes. We will slightly modify this algorithm to support the geocoding of three-dimensional geographical points instead of two-dimensional points, we will call these new codes Extended Plus Codes (EPC).

This will allow the identification of the workspace contexts of any MEL.

Finally, we intend to use as names for the MELs the EPC of the relative workspaces, and we want to provide the MELs name using the OCE-DNS.

OCE-DNS, as part of the Osmotic Infrastructure, can inform the SDMem about the workspace of the MELs inside it. Finally, this information can be exploited by the orchestrator to properly update the migration rules.

The second problem we aim to solve, using the OCE-DNS, is to enable the transparent migration of MELs inside the SDMem. To hide the MEL migration process, we need the OCE-DNS architecture meet the following requirements:

- Dynamic Record Resource (RR) database;
- Frequent updates to RR entries;
- High availability;
- Low latency;
- High performance

The updates on the RR records, if properly synchronized with the osmotic migration, will allow following the migration process of the MELs between the nodes, making this process transparent.

In Figure 6.2, we are summarizing from a very high point of view the final result of our infrastructure:

Figure 6.2: Virtual vs real MEL position

1. we identify a three-dimensional area in the World with the EPC 8FCQ5G68+69ˆ10;
2. we associate to this EPC an URL, in this case, 8FCQ5G68y69e10.osmotic;
3. we use this domain name to point to the MEL serving this area;
4. If the MEL migrates from the physical workspace to a Cloud infrastructure, the domain name follows this migration.

6.4 Design

In this section, we discuss the design of our three-dimensional geocoding algorithm that allows us to express any three-dimensional geographic area in the world through a string of variable lengths based on the area's dimensions. Generated geocodes will be used to identify the MEL workspace. We also present the design of the DNS infrastructure that meets the requirements defined in the previous sections, which include a dynamic RR database, low latency, high availability, and the ability to serve the geocodes as domain names for the MELs.

Using geocodes as names for MELs will allow the identification of the MELs themselves by third parties, such as other MELs running inside or outside the same SDMem, or clients that interact with MELs in the identified context. The MEL naming service can be done using a DNS server that serves geocodes as DNS records associated with the proper MELs. This approach allows explicitly provides clients with information about the context where a MEL works and enables the service discovery based on the geocode. Also, the OOE benefits from that since it can use the information derived by the geocoded names to adjust the migration rules according to the already considered factors, like security, network, and so on.

The just-described DNS infrastructure is the OCE-DNS.

6.4.1 Three dimensional Geo Codes

The geocoding algorithm should be able to produce geocodes that meet all the following requirements:

1. usable as DNS names;
2. code length dependent on the dimension of the referred area, to represent different size areas;
3. use of hierarchical structure;
4. representation of three-dimensional points;
5. offline working to guarantee the service even without external configurations.

We decided to start with the Open Location Code algorithm by Google, released with Apache License 2.0. The Open Location Code Algorithm creates geocodes called Plus Codes. This algorithm is based on a clear encoding of latitude and longitude information based on the WGS84 standard. The algorithm works offline and does not require any external configuration. Plus codes are generated following a Discrete Global Grid (DGG), and the output codes use an alphabet of 20 digits composed of numbers and letters. The "+" character (U+002B) is used as a non-significant character to aid formatting, it is called *formatter separator*. The "0" character (U+0030) is used as a padding character before the format separator. The minimum length for a Plus code is 2 digits (precision: 2226 km), and the maximum length allowed is 15 digits (precision 4 x 14 mm). Finally, The algorithm uses two different approaches to calculate the first ten digits and the last five. The first ten digits are produced in pairs, the algorithm recursively encodes both latitude and longitude in base 20 using the area found in the DGG during the previous iteration. For this reason, we cannot get an odd number less than 10 digits in length. The last five digits, on the other hand, are calculated one by one by encoding the latitude in base five and the longitude in base four and, again the area found in the DGG during the previous iteration. For this reason, we can calculate odd plus code only longer than 10 digits.

Open Location Code (and other well-known geocode algorithms, such as Geohash or what3words) only considers two-dimensional points because they provide geographical referencing into a plan (longitude and latitude). However, this approach doesn't allow

localizing a resource considering the altitude. This aspect is critical when we want to use geocoding in indoor and multi-floors environments, such as skyscrapers. For this reason and to increase as much as possible the granularity of our naming service, we decided to extend the geocode dimensional points to three. This information is not kept in the Plus Codes, so we need to make a small change to the Open Location Code to also encode the altitude of a certain area. Our idea is to concatenate the Plus Codes with the height expressed as altitude from sea level, as suggested by the WSG84 standard, using the "^" character as altitude-separator. Finally, we refer to this new geocode format as EPC. As explained, the Open Location Code allows you to define areas of different sizes, which allows you to target entire cities, buildings, or rooms simply by varying the precision of the algorithm. The precision on the height value is not variable, as the purpose of the EPC is to understand if an identified area is located for example, on the first or second floor of the building or at the bottom of a door rather than at its top. A precision set at 0.1 meters can be considered sufficient to have this detail. Finally, as a convention, we decided to represent the altitude from sea level in meters with one decimal digit using the dot "." character as the decimal separator. Using this convention:

- 8FCQ5HR4+V3^10 is a valid EPC;
- 8FCQ5HR4+V3^5.5 is a valid EPC;
- 8FCQ5HR4+V3^5.55 is a invalid EPC;
- 8FCQ5HR4+V3^5,55 is a invalid EPC;

6.4.2 EPC as MEL Names

The EPC allows us to identify an area served by smart services, but we also need to connect smart services to the area itself. To this aim, our idea is to use the EPC as domain names associated with the MELs that are active in the area referred to the specific domain. Using this approach, any third-party client will be able to identify the geographical workspace of a MEL only using the MEL name itself. At the same time, the client can understand if a workspace is covered by Osmotic Services just by pinging the relative EPC. Using EPC as the MEL name is also a better approach than using other information stored in a RR database, like, for example, TXT records or LOC records, because it allows one to directly associate the MEL to its workspace without any further needed queries.

From a practical point of view, EPC has to be compliant with the domain name encoding constraints, which are:

1. domain length shorter than 64 characters;
2. domain alphabet composed only of alphanumeric characters (excluded the characters "-" and ".").

We must therefore replace the characters "+" and "^", which are not allowed, we can replace them respectively with the alphabetic characters "y" and "e" that are not used in the alphabet of the Open Location Code algorithm, thus they cannot create any collision.

6.4.3 OCE-DNS Infrastructure

One of the goals of this work is to grant a dynamic Osmotic environment where MELs can transparently migrate inside an SDMem. The concept of transparent migration is critical in Osmotic Computing because it guarantees that third clients that are communicating with a MEL do not perceive a downtime for the MEL during its migration and they do not need to update their configuration to keep the communication up with the MEL when the migration has been completed.

The OOE is demanded to move the MEL inside the SDMem; the OCE-DNS, instead, is demanded to create a reference to MELs according to the migration process thus the OCE-DNS must be able to update the name refers to a MEL as soon as its migration is completed. In practical terms, the OCE-DNS must adopt the behavior described in Figure 6.3. In the figure are shown two different Osmotic nodes, respectively, with the IPv4 addresses 1.1.1.1 and 2.2.2.2. The figure also showed a MEL that offers traffic control in the area pointed out by EPC 8FCQ6H33yXFe10. For the first time, the MEL is run on the first Osmotic node. Thus the OCE-DNS associates the domain name 8FCQ6H33yXFe10.traffic.osmotic with 1.1.1.1 IPv4 address. In a second time, the MEL is completely migrated to the second Osmotic node; thus the OCE-DNS table is updated to associate the domain 8FCQ6H33yXFe10.traffic.osmotic with the IPv4 address 2.2.2.2. During the migration instead, the DNS record is unchanged, since the MEL is no more available from the departure node only when the MEL is up and running in the arrive node.

To enable the geographic naming using the just-defined EPC geocodes we need a DNS server able to read these codes. The OCE-DNS system is decoupled into an RR database that stores all the DNS records and a DNS server that resolves the DNS queries using the RR database. Decoupling the database and DNS service is an important step that needs

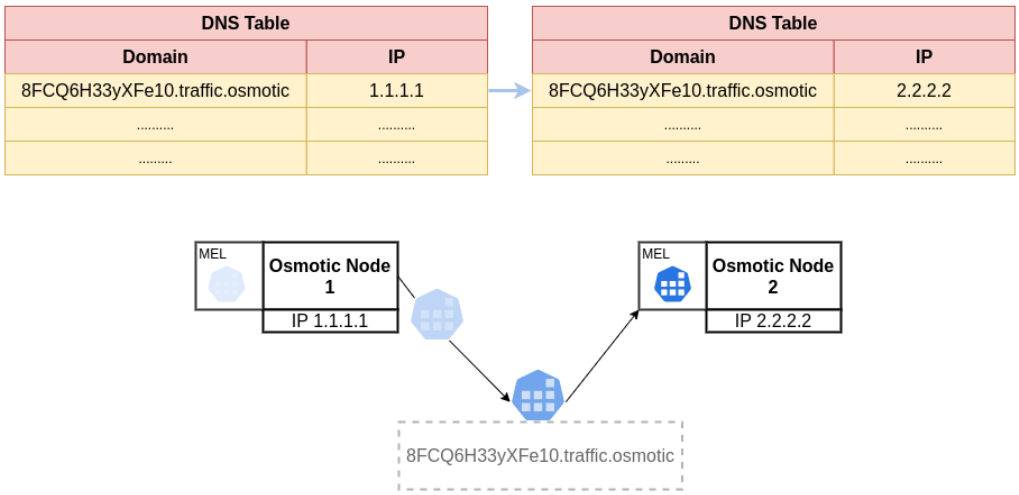


Figure 6.3: RR update

to be careful, in fact, a bad architecture or a not well-organized network topology could generate high latency during the communication with the DNS server and the RR database, this latency will be propagated to the final clients. The interaction between the MELs and the Naming system should follow two different flows. The first flow starts from an authenticated MEL that updates an RR entry exploiting the API that the RR database should expose. The second flow starts even from MEL but is directed to the DNS server to solve any DNS generic query. The DNS server, therefore, must be sensitive to any change to the RR database, it must not include any concept of data caching to serve the newest data. This characteristic requires that the TTL DNS record parameter is ignored or forced to zero. This behavior constrains the DNS server to verify every time the value of a DNS record in the RR database, avoiding any caching.

The proposed architecture is shown in Figure 6.4. In the figure, we distinguish the server side above the figure and the client side, below the figure. The server side is composed of the DNS server and the RR database, and they are deployed using a Micro Service architecture. The RR database microservice is based on a replica set that grants high availability of the data. The interaction between the RR entries and the database clients are enabled by APIs. The DNS server is a single microservice able to reply only to DNS queries and to read data from the RR database using the proper connector. Both the RR database and the DNS server are deployed using Docker containers. This approach is naturally integrated with Micro Service architecture since it is based on the concept of *Single Container Single Service*, thus granting

high granularity, horizontal scaling, and easy deployment. We identify as OCE-DNS the architecture composed of the RR database microservice and the RR database microservice.

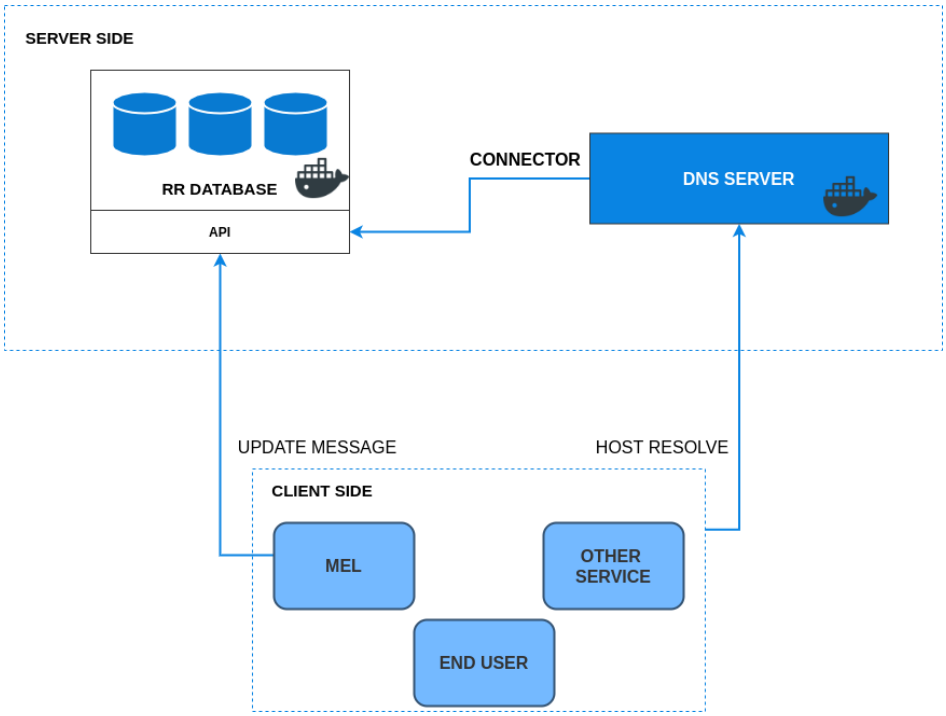


Figure 6.4: OCE-DNS architecture

Figure 6.4 depicts also the interaction between the DNS server, the RR database and third party clients. Basically, the DNS server can be queried by any external node that wants to resolve a domain name, a client can be the final user or other MELs. In any case, The DNS server will read the proper RR data in the database anyway. The update keys API are thought to be used only by the interested MEL, or by a MEL orchestrator, this depends on the installed Osmotic Infrastructure.

6.4.4 RR Types

The information that a DNS server can provide is varied and is classified according to the type of RR stored in the DNS server database.

In particular, to save all the information about the real positioning of an Osmotic node and the run position of a MEL, we intend to use the following RR types:

1. **A record**: to associate the IP address of the Edge node serving the data area to an EPC domain name;

2. **PTR record**: to enable reverse DNS, then to get from the IP address to the domain name;

3. **SRV record**: to indicate on which door the MEL we intend to contact is exposed;

4. **TXT record**: to associate the MAC address of the physical device installed in the area identified by the EPC to an EPC domain name, if it exists.

6.5 Implementation

In this section, after a brief discussion of possible enabling technologies, we discuss the OCE-DNS prototype implementation.

6.5.1 Enabling Technologies

Starting from the architecture proposed in section 6.4, we need to identify the enabling technologies required for the installation of a database for the RR, and for the DNS server capable to satisfy the OCE-DNS queries coming from the clients using the data stored in the RR database. The RR database we intend to use is based on a **Etcd** [1] cluster. Etcd is a key-value directory distributed storage. Etcd is deployed using a size variable replica master-slaves cluster. The consistency in the cluster is granted using a Raft consensus, both for writing and reading operations. Etcd allows, eventually authenticated, users, to read, write and watch new and already existing keys, through the use of gRPC or HTTP API. In our case, we are going to use the EPC as keys, and the values as the data of the RR.

The DNS server we intend to use in collaboration with the Etcd Cluster is the CoreDNS server. CoreDNS is an open-source DNS server based on the concept of plugin chains. Basically, it works like a simple DNS server that can serve RR records written in a generic data storage, as a file, or to an external DNS or a key-value data store like Etcd. CoreDNS works by defining several zones, each zone can be served using different plugin chains. Plugins can offer several services that allow them to generate a response, and communicate with other services or information useful for the next plugin in the chain. CoreDNS can be easily integrated with the Etcd cluster using the proper plugin, thus all the queries sent to the CoreDNS can be solved inside the Etcd cluster by the use of the Etcd connector plugin. CoreDNS can use the Domain Name System Security Extensions (DNSSEC) to implement network security functionalities. In our architecture, this could ensure that updates arrive only from trusted MELs.

[1] https://etcd.io

6.5.2 OCE-DNS Infrastructure Deploy

Starting from the architecture shown in Figure 6.4, we can personalize it to include the proposed enabling technologies in section 6.5.1. The result is shown in Figure 6.5, where we replaced the DNS server with the CoreDNS component, and the generic RR database server with a two-node Etcd cluster, finally APIs used to interact with the RR database have been replaced with the Etcd native gRPC and REST APIs.

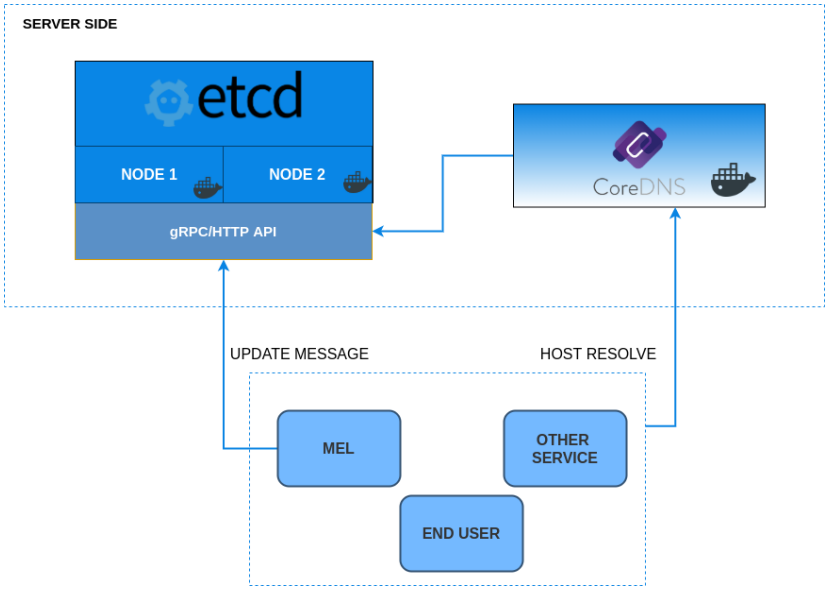


Figure 6.5: OCE-DNS infrastructure

All the components shown in Figure 6.5 are packaged inside Docker containers according to the design choices. We run a two-node Etcd cluster. This configuration is not resistant to any failure in the cluster, but it allows the splitting of the traffic between the two nodes. Such a configuration reduces the latency during the raft consensus since there are few nodes to synchronize. In the section 6.6 we will compare the behavior of the cluster in terms of performance, changing the size of the cluster.

Any node of the Etcd Cluster can be queried by passing through the default TCP port 2379, in our configuration only the CoreDNS server needs to query the Etcd Cluster so we can bind the communication to the local network.

6.5.3 RR Keys structure

The key-value couples in the Etcd cluster act as the Record Resource of the DNS server, in particular, every key is read by CoreDNS as a domain name, and the JSON formatted body contains all the information that allows generating the relative RR entries, like A records,

PTR records and SRV records.

Etcd adopts a flat key storage system starting from version 3 of the API, this means that the division into the path of the keys is purely logical and there is no physical connection between a key and its apparent father key, this happens because the concept of directory key has been removed. This approach has been adopted to improve system performance, but requires greater care in saving keys, as it could create collisions when CoreDNS accesses the keys. The main constraint that derives from this structure in key management is that it is not possible to store two keys where one is completely contained in the other, in fact, CoreDNS would return the values of the requested key and all those that contained it. These conditions are the key structure used to store the RR records in the Etcd cluster explained below.

Keys for SRV and A Records SRV and A RR are used to associate to a given domain the port number where the service is exposed and the IP of the host is pointed by the given domain. This information is strictly related since if MEL is migrating, it should update the IP address of the host where it is going to run, and the port where MEL is listening that could change during the migration. Etcd allows storing these two pieces of information in the same key. The key, in this case, is the domain name we want to associate with the MEL running on the Osmotic Node. The key structure uses the following pattern: */Osmotic/⟨ClassOfTheService|NumberOfService⟩/⟨EPC⟩*, where

- **EPC** is the Extended Plus Code that gives the workspace of the MEL;
- **ClassOfTheService** indicates the kind of the service that is running, it is used to classify and distinguish the service that is covering the same workspace;
- **NumberOfService** is used to enumerate the services of the same class that is working in the same context;
- **Osmotic** is the root domain base.

The JSON value of this key must contain the host field and the port field. The host field will be read by CoreDNS when it will receive a type A DNS query for the domain EPC.ClassOfTheService|NumberOfService.osmotic; the port field instead, will be read when the CoreDNS receives an SRV DNS query for the same domain.

Keys for PTR Records PTR records become very useful to reverse an IP address to a given domain name. This information could be used to get the services that are running in a

given host. For this kind of records, the keys must use the following pattern: */ocedns/arpa/in-addr/⟨first ip block⟩/⟨second ip block⟩/⟨third ip block⟩/⟨fourth ip block⟩*. The JSON value of this key must contain the *host* field with the domain name as the value.

Keys for TXT Records TXT record is a trick used to store the MAC address of the device that is physically installed in the area we are pointing to with the EPC in the given domain name. CoreDNS can serve as TXT RR any key under the root key where the value is a JSON with the *text* field fully filled. Unfortunately, we cannot store the same key information for SRV, A, and TXT records, because CoreDNS is not able to manage all the information together yet, so we need to use a second key pattern as follows: */osmotic/mac/ClassOfTheService|NumberOfService/EPC*. We want to remark that is not possible to put the *mac* special keyword as the last element in the key, since all the other keys will become fully included in this key, creating conflict.

6.5.4 CoreDNS configuration

CoreDNS can be configured to serve multiple zones, i.e. domains. In our case, we have to configure the .osmotic zone and the 0.0.0.0/0 zone. The first zone is for resolving all domains ending in .osmotic, the second is for resolving reverse DNS PTR queries. Currently, we set that all the IP addresses must be searched within Etcd, but if we already knew that Osmotic nodes belong to a given subnet we could configure the Reverse DNS in a more precise way.

Both zones can follow the same CoreDNS plugin chain, of which the most important is the Etcd plugin. The latter requests input the URLs of the Etcd cluster nodes to query the DNS record search. In the plugin chain, we deliberately avoid the use of the cache plugin, as suggested by the CoreDNS template configuration. This allows us not consider the TTL value in the RR database, forcing the CoreDNS to check every time the value of a DNS record in the Etcd cluster. The example configuration is shown in the code 6.1.

```
1 osmotic 0.0.0.0/0 {
2   etcd {
3     path /ocedns
4     endpoint
5     HTTP://node1.myetcd:2379,
6     HTTP://node2.myetcd:2379,
7   }
8 }
```

Listing 6.1: CoreDNS plugin chain

6.6 Performance Evaluation

OCE-DNS provides a dynamic system that allows updating RR entries on the Etcd cluster. Specifically, MELs use Etcd APIs to logically remap them into OCE-DNS. This mechanism has been designed to enable the transparent migration of MELs.

In this Section, we analyze the performance of the Etcd cluster on which our OCE-DNS is based. In particular, we will analyze the Average Time to Respond (TTR) value of a Read or Write request sent by the MELs. Experiments were performed considering different Osmotic Computing clusters: cluster 1 including two nodes and cluster 2 including five nodes. Cluster 1 is not capable of resisting the failure of one node, whereas Cluster 2 can tolerate failures of up to two nodes. In the end, all the EPC contained in the requests are 21 digits long (15 for the Plus Code, 6 for the height). This value is the worst realistic case since we are considering the maximum length for the Plus Code part and the maximum altitude equal to 9999,99 meters.

6.6.1 Testbed Setup

Our testbed setup has been chosen to fit different possible environments where Continuum is involved. As shown in Figure 6.6, we use the *c* parameter to set the number of clients and the *r* parameter to set the number of requests for each client. The *s* parameter is the size of the Etcd cluster that we already said equal is to two or five.

The parameter *c* can take the following values: 10, 100, 1.000. Of course, a reader client could be any host querying the DNS server, for example, to resolve the domain name of a MEL, instead, a writer client can be only a MEL that is authorized to directly update its DNS record.

Let's imagine that each client uses one MEL. In this case, each value of *c* represents a specific Osmotic environment. The smallest environment with only 10 MELs could be a basic environmental safety analyzer for a small street that only checks whether certain environmental parameters such as temperature or air quality reach dangerous values. The average environment with 100 MEL could cover an area larger than a street, such as a square, in this case, we might want to be interested in inspecting other important factors such as the aforementioned camera controls, in this scenario the analyzer service video could be run in parallel and then distributed across several MELs. In the end, the largest environment with 1000 MELs could be located in a park, where it is necessary to monitor environmental parameters, to control the entire area with several smart cameras and related services, and many web services implemented to offer specific services to visitors.

The *r* parameter can take also the values 10, 100, and 1000 per node request. The number of requests per node represents the dynamism of the environment, which could change over time, depending on the nature of the environment. For example, the square Osmotic Environment could be more stressed during the night when people are hanging around, instead, the park Osmotic Environment could reach the highest traffic value during the day of the weekend.

The lightest configuration includes 10 MELs each one sending 10 requests, for a total of 100 requests, whereas, the heaviest configuration includes 1000 clients each one sending 1000 requests, for a total of 1.000.000 requests. As we see in Figure 6.6, each of the *c* clients has

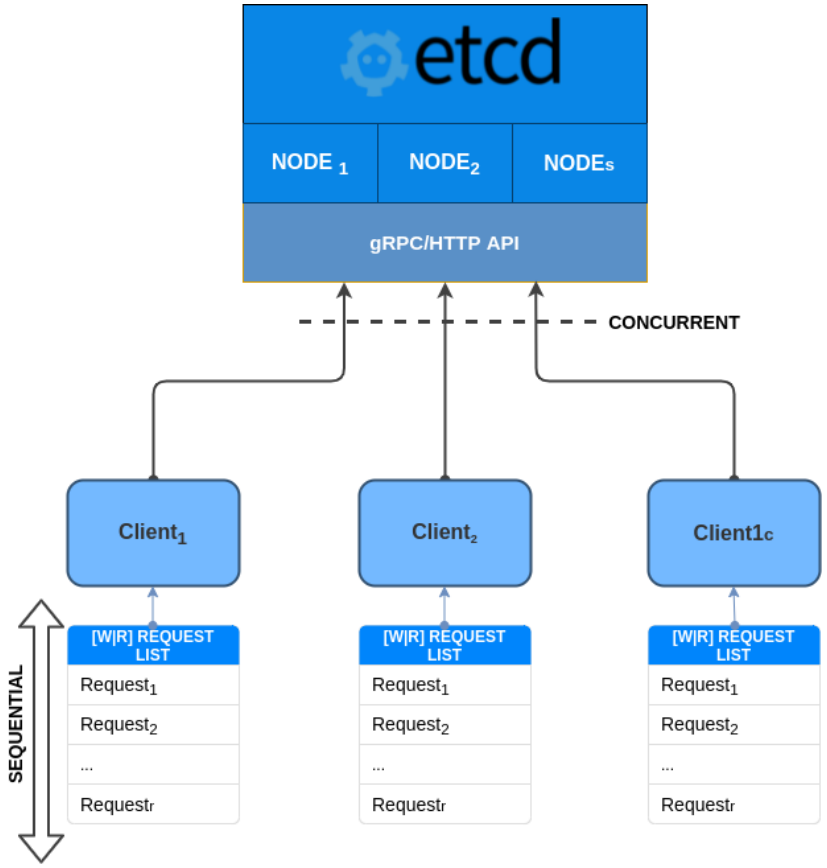


Figure 6.6: OCE-DNS experiment setup

a different requests list and in the list, all the requests belong to the same type of operation (writing or reading). The Clients pop a request from their own list one by one, sequentially as soon as the previous request has received a response. Clients are not synchronized between them instead, for this reason, the Etcd cluster could receive at most *c* parallel requests at the same time.

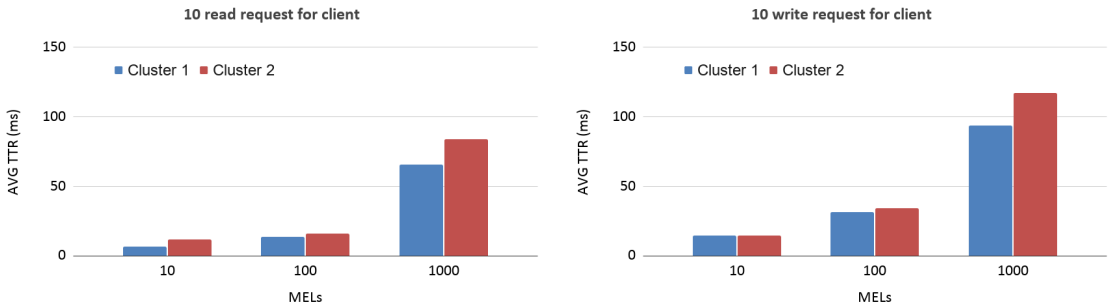
In the tests, we carried on we separated the writing requests from the read requests, and we compared the behavior of cluster 1 and cluster 2, varying the kind of operation, the

number of the clients c, and the number of requests for the client r.

Figures 6.7, 6.8, 6.9 show the average TTR related to read/write tasks obtained in clusters 1 and 2, respectively considering 10, 100, and 1000 MELs requests. The Etcd Cluster nodes were deployed inside Docker containers running in different VMs instantiated in the same region of the OpenStack environment. Their characteristics are summarized in table 6.1.

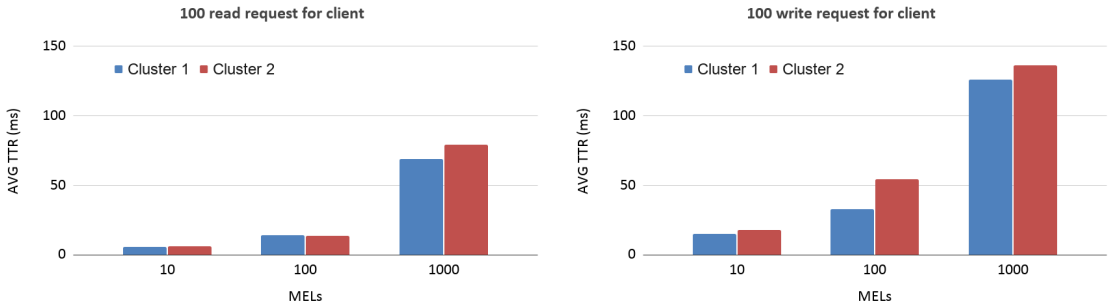
Parameter	Values
CPU	3.1 GHz Intel Xeon 4 cores
RAM	4GB
Disk	50GB, r: 3.0 GB/s, w:220 MB/s
OS	Debian 10

Table 6.1: OCE-DNS testbed setup



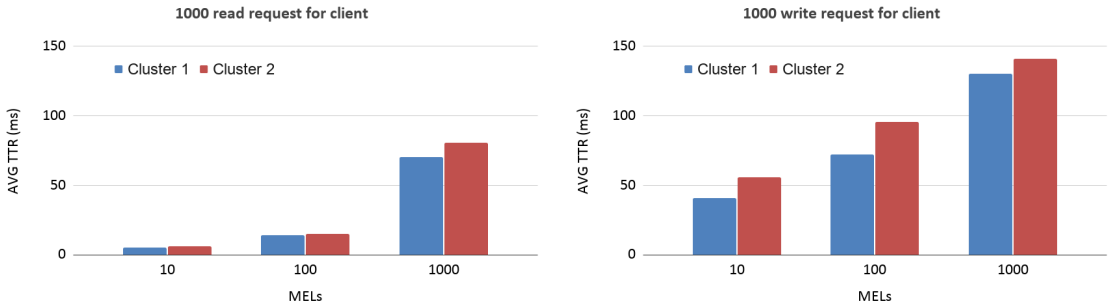
(a) 10 read per MEL **(b)** 10 write per MEL

Figure 6.7: 10 DNS requests per MEL



(a) 100 read per MEL. **(b)** 100 write per MEL.

Figure 6.8: 100 DNS requests per MEL



(a) 1000 DNS read per MEL.

(b) 1000 DNS write per MEL

Figure 6.9: 1000 DNS requests per MEL.

From our experiments, we can observe that regardless of the type of operation sent to the Etcd cluster, an osmotic environment including a small number of nodes is capable of responding in a shorter average time than a cluster arranged considering a greater number of nodes.

In all the histograms, especially in the ones shown in Figures 6.7b, 6.8b and 6.9b where only write operations are considered, we can observe how the delta between the average TTR of cluster 1 and 2 increases augmenting the size of the Osmotic environment and thus the number of requests per time, that arrive in the cluster. This behavior is more evident in the Osmotic environment composed of 100 MELs, in fact, cluster 2 performances, with 1000 MELs inside and with the maximum number of requests per MEL, are slightly worse than cluster 1 in the biggest osmotic environments. In Figures 6.7a, 6.8a and 6.9a, where only read requests are sent to the clusters, the delta between the average TTR of cluster 1 and 2 is less evident, in particular with 10 or 100 MELs in the osmotic environment the performances are very similar. The reasons that justify the difference in behavior in the Etcd cluster during the write and read tasks are closely related to the hardware characteristics of nodes. Etcd uses a rafting algorithm both during the reading and writing operations. When a read is performed on a node then, the read value is compared with the read value in the other nodes, and the read value that reaches the consensus quorum is sent to the client. During a write request, on the other hand, a positive response from the server is obtained only after the quorum of the nodes has committed the write itself. But, as reported in table 6.1 The hard disk mounted in VMs guarantees performance around 10 times better for reads than for writes operations. This means that in the most stressful environments, the requests tend to overlap more, increasing disk usage and degrading overall cluster performance. In smaller clusters the consensus is composed of fewer nodes, thus the requests can be satisfied better, stressing lower the cluster, finally granting good performances. Read operations, finally, are consumed

very fast, avoiding any overlap and allowing the system to reply faster to any situation.

6.6.2 Discussion

Conducted experiments allowed us to make a series of important considerations on the behavior of OCE-DNS. Going into the specifics of cluster response times, write operations, in the worst case, are performed in 0.14 seconds. This value is added to the time needed to complete a MEL migration. As we discussed in Section 6.2, the currently available solution of MDNS usually takes 10 seconds to propagate the resource migration information. The latter includes the migration time of the data used by the MEL and the initialization of the new MEL in the destination node. The OCE-DNS overhead that we have to consider during a MEL migration is therefore minimal and does not significantly affect the execution time of the migration itself. The maximum TTR value recorded allows understanding also which is the maximum rate that the system can withstand, in this case, one migration every 0.15 seconds. Several works demonstrated that on average a container bootup in a few seconds [129], [130] the TTR we measured in our system negligible compared to this value, therefore, makes the OCE-DNS suitable for such environments.

The reading times in an Etcd cluster, on the other hand, allowed us to understand how expensive the resolution of the name of a MEL with its real address is. In the worst-case scenario, considering the heaviest test with 1000 MELs and 1000 requests for each MEL, the average response time remained below 0.09 seconds. Also, in this case, the value obtained is an excellent value since a generic interaction via the network is concluded with a time that is around the order of a second, therefore, the address resolution time takes up only 1/100 of the time of the entire communication. In the end, these values have been calculated considering the heaviest data transmission case with the longest available domains, thus we obtained results that describe time delays that we do not expect to overcome in a real use case.

Another interesting parameter concerns the comparability of performance between an Etcd clusters 1 and 2. Although the second one degrades the performance of the entire OCE-DNS system, the average response times remain comparable to cluster 1. However, the fault tolerance offered by cluster 2 makes a higher latency acceptable. Thanks to the consensus protocol adopted by Etcd in fact, cluster 2 would remain functional and available for the osmotic infrastructure even if two nodes become unreachable for some reason. Instead of considering cluster 1, on the other hand, the entire operation of Etcd would be unavailable as long as one of the two nodes becomes unavailable. Therefore, we can assert that the Etcd

infrastructure on the basis of OCE-DNS is solid and performing enough to be used in an osmotic computing environment.

6.7 Conclusion

In this Chapter, we deal with several issues related to Continuum. The first one is the service naming, which is the ability to identify a service in a local or public network. The second one is identifying the geographical context where a service operates. The third issue is hiding a service migration over the Continuum infrastructure to clients. We applied the Osmotic Computing paradigm to address this issue, but we extended it to include a powerful DNS called OCE-DNS used to discover and list services using their position as key. Our geocoding algorithm, called EPC, is built on Google's Open Location Code. It can identify any three-dimensional and variable-sized space in the world quickly and easily. The EPC was used for identifying the geographical workspaces in which the MELs provide services. Furthermore, the EPC has been manipulated to be used as domain names, served by the OCE-DNS infrastructure, which allowed us to enable the MELs naming and provide new inputs to the OOE to define intelligent migration rules. We built the OCE-DNS infrastructure, a DNS system able to serve the EPC as names for the MELs. Using a very performing architecture, we achieved quick DNS reading and updates in order to manage the transparent Osmotic MEL migration.

CHAPTER 7

Orchestrating Applications in the Continuum

Computing at the Continuum implicitly includes the concept of migration, that is the ability to move a computation from one infrastructure to another one that is more suitable for that service at that moment to guarantee the continuity of service or a good QoS. We have seen that this can be achieved using an orchestrator like Kubernetes, but in some constrained and or huge environments, this kind of system could fail. This chapter proposes Tolerancer, a micro-orchestrator composed of distributed components that continuously interact in a peer-to-peer fashion aiming at detecting stress situations or node failures. Then, it makes decisions to avoid or solve any potential system failures. The performance evaluation of Tolerance, using a real testbed, shows that it can efficiently ensure the needed level of fault tolerance.

7.1 Introduction

A typical continuum scenario is industrial manufacturing; in this kind of competitive environment, manufacturers need to enhance their infrastructure to increase revenue. Thus, they adopted the cloud manufacturing model as it offers an encapsulated variety of manufacturing resources as services to meet customers' demands with lower costs and better performance. However, cloud manufacturing, in its classical architecture, has some limitations. The architecture of the cloud is of a centralized fashion that assumes stable connectivity to offer convenient services. But uninterrupted connection cannot be guaranteed. At the

same time, the Industrial Internet of Things (IIoT) applications must work even when the connection is temporarily unavailable or under degraded conditions. In addition, cloud computing assumes that there is enough bandwidth to transfer data between the physical location of the manufacturers' devices and the cloud data centers, which is also not guaranteed. Moreover, transferring massive data results in network bottlenecks and leads to latency issues for applications [131], and this may cause a deterioration in computing performance. Such limitations in the cloud layer may result in system failures. Thus, the manufacturers utilize the edge computing model to complement the cloud by decentralizing the computing and storage resources and moving them closer to the plants and factories, aiming to improve service quality.

However, systems may also fail at the edge layer, mainly due to the low scalability and limited resource capacity at this layer [132]. What makes providing reliable and fault-tolerant services in manufacturing environments more complex is the relationships among manufacturers. The manufacturers need different types of services as the life cycle of their product development comprises different stages. The products' dynamic, complex, and long life-cycle processes may result in service failure [133]. Thus, there is a need to manage the failure that may occur to cloud services offered to the manufacturing and industrial sectors. Without proper fault tolerance approaches, multiple manufacturing services will fail to lead to great losses.

This chapter aims to investigate service failure and overloading in cloud-edge continuum environments, using the manufacturing a use case where the continuity of computing is critical. More precisely, this work is trying to answer the following research questions:

(1) How to design a robust fault tolerance approach to avoid and/or deal with any possible failure in the nodes that host IIoT applications?

(2) How can a hybrid (Proactive/Reactive) fault tolerance approach be designed in edge-cloud manufacturing environments?

7.2 Related work

This section presents the related works. Most existing fault-tolerance approaches can be classified into two categories: proactive approaches to avoid the expense of system fault by predicting it in advance and reacting accordingly, and reactive approaches to handle the system's fault after it happens by utilizing adequate techniques. However, the literature includes a few hybrid approaches.

There are several related existing proactive approaches. In [134], the authors proposed a proactive fault tolerance approach to prevent system faults within the federated cloud environment. The environment is modeled as a multi-objective optimization problem that maximizes the profit and minimizes the VMs migration cost. The approach can re-distribute VM from faulty providers to non-faulty ones within the federation. However, this work considered applications that are served by VMs at the cloud layer and did not consider the features of the edge nodes. In [135], the authors proposed a fault-tolerant approach to maintain system availability. The approach includes the following components: fault manager, controller, and load balancer which work together to ensure a fault-tolerant environment via redundancy, optimized selection, and checkpointing. The work in [136] presented an approach that models the temperature of the CPUs in a virtualized cluster to expect a potential failure in a specific physical machine (PM), and, accordingly, migrates VMs from the detected PM to be hosted on another PM. The selection of the new PM is represented and solved as an optimization problem. However, this work targeted VMs in the cloud environment, not containers at the edge. In [137], the authors proposed a fault-tolerant approach to work in the fog layer. The approach utilizes the checkpointing technique, and at the same time, it applies load balancing based on Bayesian classification to consider the energy efficiency of the fog devices. However, the approach was not evaluated in a real testbed. The work in [138] presented a preemptive migration prediction model, called PreGAN, to detect and classify faults in edge computing environments. PreGAN can migrate services from one node to another based on the features of the potential detected failure.

On the other hand, there are related reactive approaches. In [139], the authors presented a two-stage fault tolerance approach (off-line and online) to improve the reliability of the manufacturing network. The off-line stage ranks the manufacturing services according to their importance in fault tolerance, then the critical services are replicated. While the online stage performs a heuristic algorithm for replacing the failed services. The work in [140] presented a three-layer approach to solving the problem of system failure in cloud-edge environments. The three layers (Application Isolation, Data Transport, and Multi-cluster Management) work together to re-schedule failed processes on other available nodes. In [141], a fault-tolerant approach for recovering the failed IoT edge applications is presented. It manages and re-configures container-based IoT software in a reliable way upon software failure detection. However, the authors stated that the approach is unsuitable for low-powered devices. The authors in [142] leveraged both Primary-Backup (PB) fault-tolerant and Deep-Q-learning-Network (DQN) techniques to ensure safe execution for the edge services. However, the

approach was not evaluated in a real environment.

There are also a few hybrid approaches, combining both reactive and proactive approaches. In [143], the authors presented a hybrid model to take fault tolerance actions: proactive actions after predicting the failure probability, and reactive actions that employ replication and checkpointing techniques. The work in [144] presented a fault-tolerance approach that utilizes two directions: the first is performing a VM migration based on a failure prediction technique, and the second is by doing VM checkpointing.

To our knowledge, our work is the first to present a hybrid fault tolerance approach in cloud manufacturing environment.

7.3 System model

The system model targets the hierarchical edge-cloud continuum computing architecture (Manufacturing Environment) to prevent and/or manage the potential system failures in such environments. The system is divided into three different layers: Manufacturing layer, Edge layer, and Cloud layer, as shown in Figure 7.1, and it includes N_T heterogeneous nodes that are prone to failure, which are distributed on the edge (N_E) and the cloud (N_C) layers, such that $N_T = N_E + N_C$.

As many manufacturers prefer to process their data on-site (mainly for security reasons), this work investigates system failure in the edge layer. The nodes at the edge layer N_E can host VMs and/or containers. Containers offer a lightweight, portable, and high-performance virtual entity compared to VMs. In addition, the size of container images is smaller than VM images. This is better to be adopted in the constrained devices at the edge layer and also makes applications launch faster than VM-based applications [145, 146].

The manufacturing environment is heterogeneous. It includes many edge devices, installed at different times, with different configurations and operating systems. More devices could be added and integrated into the system anytime. Some systems adopt a *single-master multi-workers* architecture to maintain load balancing and high availability. Such systems are easier to manage compared to full peer-to-peer systems. But at the same time, as their management depends on a single master node, they come with a major issue: the potential single point of failure and, consequently, a high failure rate.

Tolerancer approach, which works within the manufacturing environment, aims at avoiding any probable single point of failure. To do so, each node in the system model has the same role in monitoring and taking fault tolerance actions. Following this fully distributed

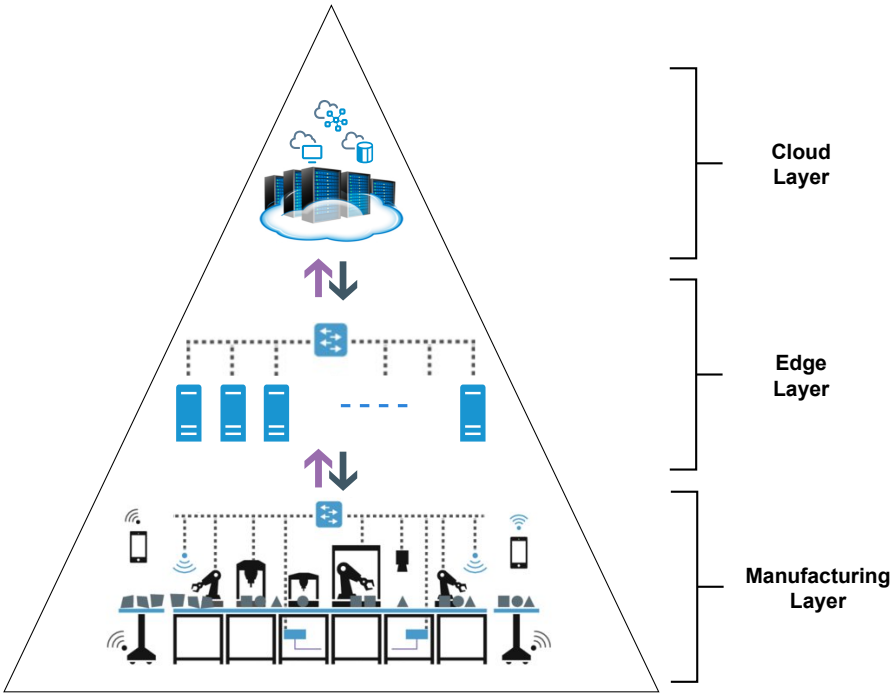


Figure 7.1: Tolerancer system model

peer-to-peer architecture, where each node $i \in N_E$ is connected with the other nodes, we designed an approach that can be hosted and run on all edge devices. The approach's components are light entities, so the nodes with low and medium computational capabilities (edge nodes) can host them.

Regardless of the device's type, capabilities, configuration, or operating system, Tolerancer can be run on it if the device can run Dockers. Docker, and any general full container-based approach, is a perfect option to be considered in the targeted environment because Docker can encapsulate the system components and run on different hardware and operating systems. Containerization helps hide such differences, automatically fetching and deploying containers on the nodes. A federation is created by connecting each device with the other devices at the edge layer. The federation's members are the edge devices that can be configured at the federation bootstrap or at the run-time. When the federation is ready, the containers hosted on the edge devices are monitored, and the statuses of the devices are observed. Each member in the federation is responsible for the management of itself (no master node to be recognized in the federation). The member is also responsible for communicating and exchanging data with the other members in the federation. This way, we can avoid the single (or n-points) point of failure situations. The events that Tolerancer can monitor include (1) high resource stressing (overloaded), (2) service down, and (3) device off.

The IIoT applications or services are deployed as Docker containers. The Tolerancer tries

to keep these services available anytime. When one or more of the previous events occur, Tolerancer triggers fault tolerance actions by involving the related or the other peers in the federation. These actions could be Proactive and/or Reactive actions The Tolerancer approach comprises three light key units: Middleware Unit (MidU), Monitoring Unit (MonU), and Planning Unit (PlaU). Each node $i \in N_E$ hosts these units which are collaborating with each other to avoid system failures and resolve them upon the failure detection. In other words, it is a proactive/reactive approach. The Tolerancer monitors the system periodically according to a predefined cycle, and the cycle interval is variable so it can be tuned based on the system status. The units are described as follows (Refer to Figure 7.2):

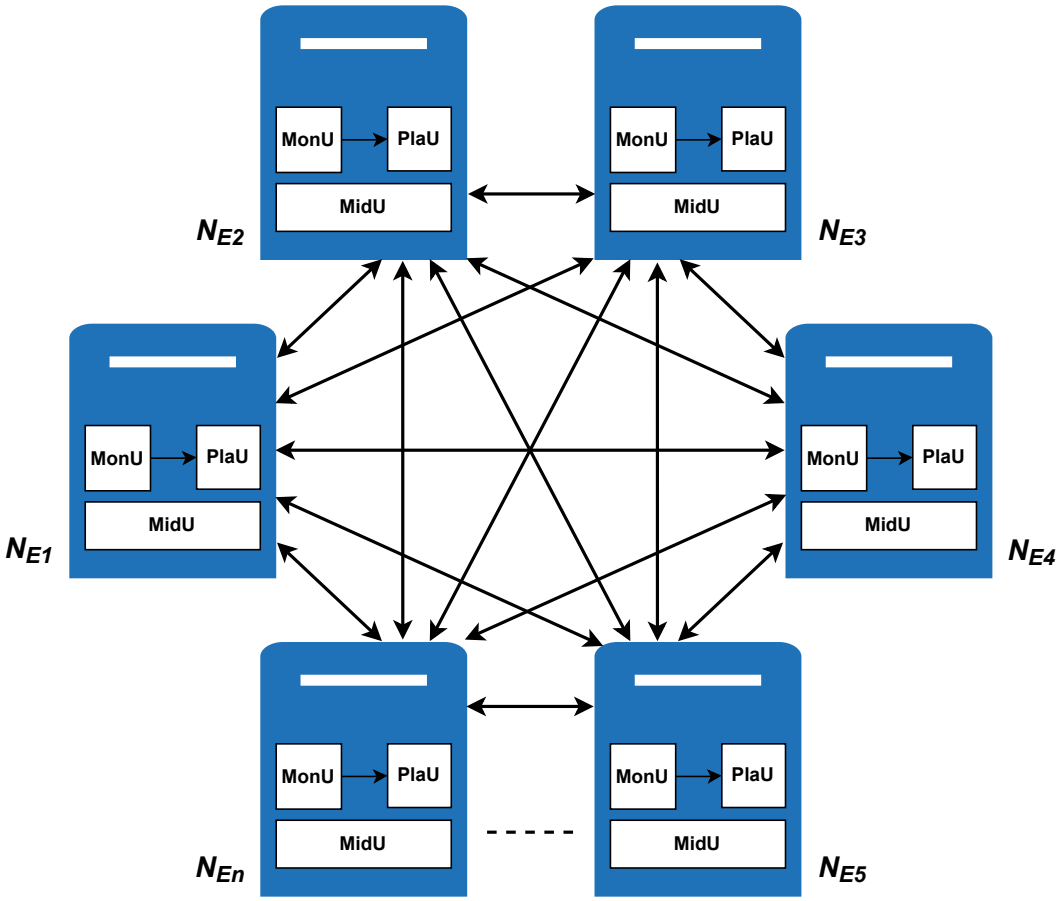


Figure 7.2: Tolerancer node connections

7.3.1 Middleware Unit (MidU)

This unit is composed of two components: the MESSAGE-BROKER and the SHARED-MEMORY. Both components are deployed in cluster mode where each edge device runs a single instance of both components.

- MESSAGE-BROKER: It is used to exchange messages between the nodes at the edge layer. To exchange information, each node employs its own MESSAGE-BROKER to send messages to the other peers. It is needed to guarantee that the system works properly.

- SHARED-MEMORY: This component is used to store all the information generated by the federation, and to make it accessed by all nodes $\in N_E$. SHARED-MEMORY stores information about the federation in general and about the nodes themselves. For example, the number of edge nodes and devices, their IDs, their health information, the running containers they host, and their migration processes.

All this information is generated and used by the Monitoring Unit's components.

7.3.2 Monitoring Unit (MonU)

The main functions of this unit are collecting data about system status and analyzing the collected data. To achieve these functions, MonU includes components that allow logging and monitor the services running on the edge nodes. The components are LOGGER and ANALYZER.

- LOGGER: it logs the status of the system and put the information into a written record periodically, based on a predefined interval. LOGGER records the following: (1) CPU usage, (2) Memory usage, (3) container status (running or failed) which is a SW-related failure that depends on the application itself, and (4) the device status (on or off) which is a HW-related failure.

  All the data is stored in the SHARED-MEMORY component to be available to all nodes in the federation.

- ANALYZER: It analyzes the data stored in the SHARED-MEMORY collected by LOGGER. ANALYZER checks the data related to each peer individually, and also the whole system status. When ANALYZER notices any cautionary data that may result in system failure, it alerts the PlaU to take action (proactive/reactive reaction). The cautionary situation can result in the following cases:

  - Case 1: over-utilized CPU and/or memory.

  - Case 2: container with a failed state.

  - Case 3: device with an off state.

7.3.3 Planning Unit (PlaU)

PlaU uses the analysis resulting from MonU (The resulting three cases), and, accordingly, performs fault tolerance action(s). Case 1 necessitates a proactive action to maintain reliability and avoid potential failure, while Case 2 and Case 3 necessitate a reactive action as the failure already happened. PlaU includes two components they are SCHEDULER and MIGRATOR.

- SCHEDULER: It is responsible for specifying the following operations when migration is needed: the service(s) to be migrated, the source device(s) that hosts the service(s), and the destination device(s) to host the migrated service(s).

  The SCHEDULER uses the data stored in the SHARED-MEMORY to take the decisions. It can employ different scheduling algorithms like Round Robin or even more complex ones.

- MIGRATOR: It receives three parameters as input: The IDs of the source devices, the IDs of the destination devices, and a list of services to be migrated. The MIGRATORs on the source and destination devices collaborate with each other to perform the migration process. After migration, the services run on the destination node and are removed from the source node. If the migration process does not perform correctly, the MONITOR can notice this in the next monitoring cycle to find a new destination. The new system status is stored in the SHARED-MEMORY components of all nodes, so the nodes in the system will be aware of what is the status resulted after the migration process.

7.3.4 Tolerancer description

This section describes the communication and the main processes involved in Tolerancer using some high-level pseudo-codes.

Communication

Communication inside the Tolerancer system is done using a special distributed Message Oriented Middleware (MoM). This MoM is implemented using RabbitMQ, a very popular open-source project that in turn, implements the Advanced Message Queuing Protocol (AMQP). The AMQP is similar to a Publish/Subscribe protocol, where producers push messages in a queue distinguished by a key called “Topic", and consumers connected to the same queue read them. AMQP adds a fourth element called "Exchange" which, in a nutshell, ensures that every message arrives at the destination, finally guaranteeing a high

QoS. These characteristics ensure stable communication among the Tolerancer 's nodes and good reliability for the entire system.

Processes

The main processes in Tolerancer are Logging, Analyzing, Scheduling, and Migration.

The LOGGER works as described in Algorithm 1. It reads information from the nodes of the system, aiming at maintaining a stable and balanced cluster. The algorithm takes a specific node as an input and outputs a stored and shared status about that node. The algorithm is activated for all nodes periodically based on a predefined period. The information is collected using an API as described in line 3, and then, in line 4 this information is stored in the SHARED-MEMORY together with the current timestamp.

Algorithm 1: The LOGGER Algorithm.

Input: The ID of the node N_{id}
Output: The node's status information
1 **Begin**
2 **While** *True*
3 $x \leftarrow get\_system\_info()$
4 $SHARED\_MEMORY.store\_node\_info(N_{id}, x, get\_current\_timestamp())$
5 **End**

The ANALYZER works in two phases. The first phase is described in Algorithm 2. Block 2-15 shows that the algorithm works over all nodes in the federation. For each node, it tries to get exclusive access using the distributed semaphore that is managed by the SHARED-MEMORY. Then, the SHARED-MEMORY gives the status information recorded by the LOGGER of the same node to the ANALYZER. The ANALYZER in step 8 examines the collected information to understand if the node is working or not. If the node is not working, The ANALYZER updates the node's status to FAILED in step 9. The algorithm is also considering the case when the node is working but the LOGGER is not updating the node's information for a while. In such a case, the ANALYZER will try to contact the node, and if no response, it will set its status as FAILED in step 12.

The second phase is described in Algorithm 3. The ANALYZER obtains a list of the failed nodes in the federation from the SHARED-MEMORY in step 3. For each of the failed nodes, the algorithm in step 6 tries to access the node's information using the shared semaphore, and in step 7, it gets the list of the container(s) hosted on the failed node.Then, in blocks 8-14, the algorithm reschedules each container to be hosted on a healthy node based on a specific scheduling algorithm. In this work, we randomly pick the new destination host among the

Algorithm 2: Analyzer Algorithm Part 1.

```
   Input: Set N_E of the nodes in the federation.
   Output: Analyzed system status, the updated nodes' status
1  Begin
2      ForEach Node_i ∈ N_E
3          try:
4              SHARED_MEMORY.acquire_node_semaphore(Node_i)
5              curr_time ← get_current_timestamp()
6              state ← SHARED_MEMORY.get_node_info(Node_i)
7              validate_state ← is_healthy(state)
8              If validate_state = False
9                  SHARED_MEMORY.set_node_state(Node_i, "FAILED", curr_time)
10             Else If state.timestamp + INTERVAL_CHECK < curr_time
11                 If try_contact(node) = False
12                     SHARED_MEMORY.set_node_state(Node_i, "FAILED", curr_time)
13         catch:
14             continue                          ▷ If the semaphore is held by other nodes, simply skip
15         end
16 End
```

healthy nodes (step 11). The migration is done through the MIGRATOR API as it initializes a transaction for accepting the new container, as described in the Figure 7.3. The Scheduling process is repeated in blocks 9-13 until no more containers are to be rescheduled.

Algorithm 3: Analyzer Algorithm Part 2.

```
   Input: Set N_E of the nodes in the federation.
   Output: Analyzed system status, the updated nodes' status
1  Begin
2      While True
3          failed_nodes ← get_failed_nodes
4          ForEach Node_i ∈ failed_nodes
5              try:
6                  SHARED_MEMORY.acquire_node_semaphore(Node_i)
7                  containers ← SHARED_MEMORY.get_containers_in_node(Node_i)
8                  ForEach container ∈ containers
9                      Repeat
10                         ▷ SCHEDULER process that randomly picks a healthy node for hosting the
                           container
11                         destination_node ← random_choice(Nodes − failed_nodes)
12                         success ← MIGRATOR.send_request(Node_i, destination_node, container)
13                     Until success = False;
14             catch:
15                 continue                      ▷ If the semaphore is held by other nodes, simply skip
16             end
17 End
```

The MIGRATOR described in Algorithm 4 is composed of two functions: *send_request* and *receive_request*.

The *send_request* function is invoked by the ANALYZER and used to ask the destination node (*destination_node*) to host the *container* that was previously hosted in a failed source node (*src_node*). This function needs to use the MESSAGE-BROKER's APIs to send a migra-

tion request to the message queue of the *destination_node* under the topic */migration_request*, as shown in step 3. There is an identifier for each migration request, we refer to it as migration ID, and it is generated in Step 3. Then, in step 4, the function will use it through the Message Broker's APIs to wait for the request's response. If the request receives a *SUCCESS* response, it means that the container is hosted in the *destination_node*. After that, in step 6, the function updates the SHARED-MEMORY with the new host of the container.

The *receive_request* function receives the requests. In step 9, the function waits for an incoming migration request message in its queue under the topic */migration_request*. When the message arrives, the function gets the faulty *src_node*, the container ID hosted on it, and the migration ID. In step 10, the function uses the SHARED-MEMORY for getting all information about the container (*i.e.*, the configuration), then it uses this information for verifying that the container is compatible with the node. If the container is reported as compatible, the function extracts the command needed for running the container and executes it in line 13. When it finishes, it sends a success message to the *src_node* message queue under the */migration_request* topic using the migration identifier as a parameter. The

Algorithm 4: Migrator Algorithm

Input: Set N_E of the nodes in the federation.
Output: New container-to-host placement, The updated nodes' status

```
1  Begin
2      Function send_request(src_node, destination_node, container): bool
3          migration_id ←
             MESSAGE_BROKER.publish(destination_node, "/migration_request", src_node, container)
4          response ← wait MESSAGE_BROKER.listen(Nid, "/migration_request/ < migration_id > ")
5          If response = True
6              SHARED_MEMORY.update_container_map(src_node, destination_node, container)
7      Function receive_request(): bool
8          While True
9              src_node, container, migration_id ←wait
                 MESSAGE_BROKER.listen(Nid, "/migration_request")
10             container_info ← SHARED_MEMORY.get_container_info(container)
11             If is_compatible(src_node, container)
12                 run_command ← container_info.run
13                 execute_run(run_command)
14                 MESSAGE_BROKER.publish(src_node, "/migration_request/ < migration_id >
                     ", "SUCCESS")
15 End
```

message exchange process between a node that is analyzing another failed node and a node that may host a new container is described in Figure 7.3.

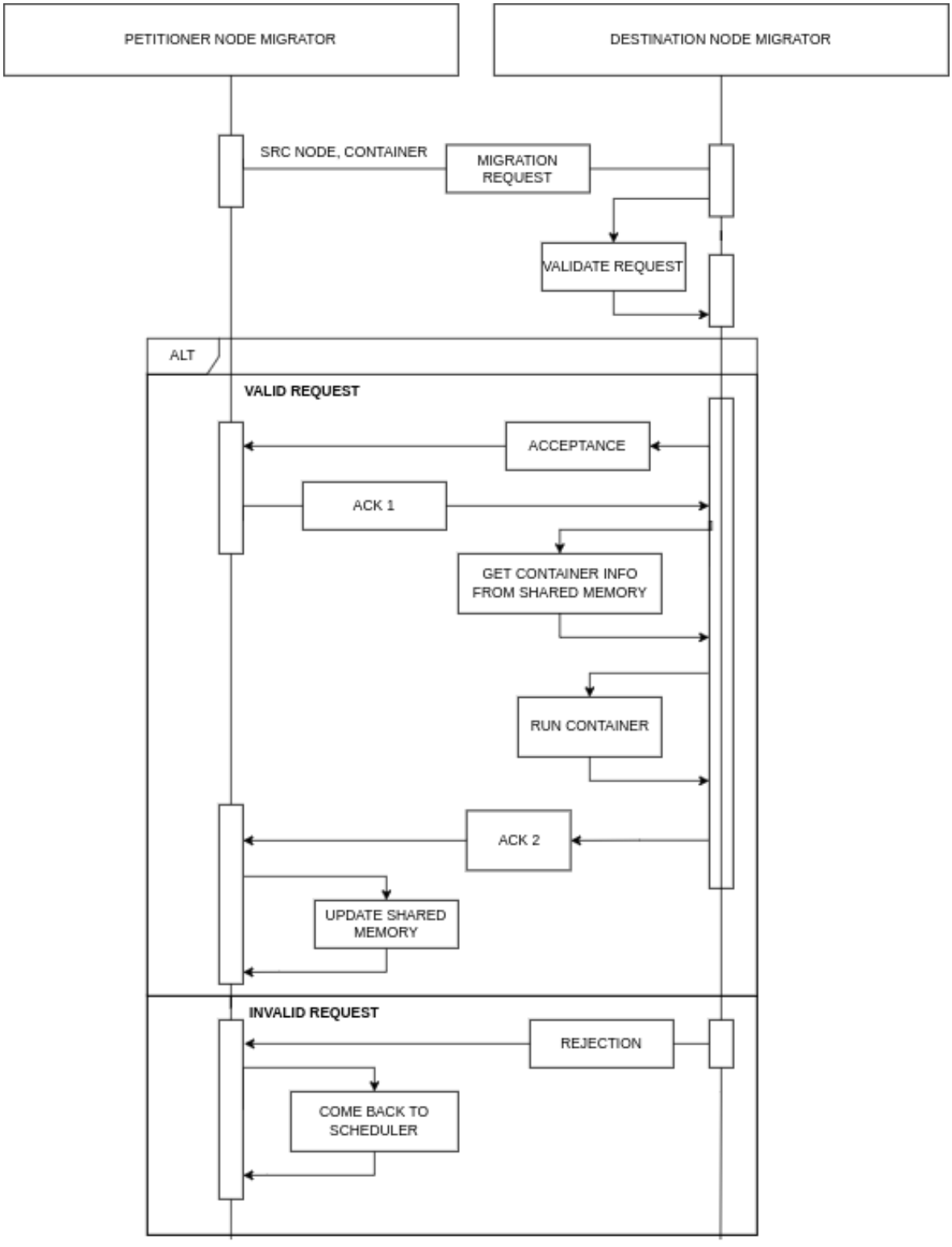


Figure 7.3: Tolerancer's Migrator message exchange

7.4 Performance evaluation

This section evaluates Tolerancer 's ability to recover the faults and move all services deployed in the failed node to another node in the edge layer.

7.4.1 Testbed and experiments

Our evaluation testbed is a cluster of nodes that represents the edge layer for a specific manufacturer. We used five nodes: four Raspberry Pi4 and one Nvidia Jetson Nano. The cluster information is summarized in Table 7.1. To evaluate the proposed approach, we targeted an edge cluster, deployed services as containers, run the containers, caused a failure in a specific node (or nodes) of the cluster by disconnecting it (or them) from the network, and then examined Tolerancer ability to recover the faults. The examination is done by monitoring the capability of the other active nodes to notice the failure and start moving all containers hosted on the failed node to another healthy one. We performed three experiments. We consider a different cluster in each experiment, as shown in Table 7.2. The cluster in experiment 1 consists of 5 nodes, the cluster in experiment 2 consists of 4 nodes, and the cluster in experiment 3 consists of 3 nodes. With every experiment, we run a different number of containers (m), each container can run Nginx web servers, and then, we cause a failure in one node. We considered the failed node hosts and run a different number of containers as follows: 3, 30, 60, 90, and 120 containers. Then, we checked if the containers on the failed node migrated to another healthy node. In addition, we calculated the time to detect the failure and the time to restore all the containers hosted in the failed node.

Name	Node	CPU	Memory
Edge1	Nvidia Jeston Nano	4-core (ARM v8) 64-bit SoC 2 GHz	4 GB
Edge2	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	4 GB
Edge3	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	4 GB
Edge4	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	8 GB
Edge5	Raspberry Pi 4	4-core (ARM v8) 64-bit SoC 1.5 GHz	8 GB

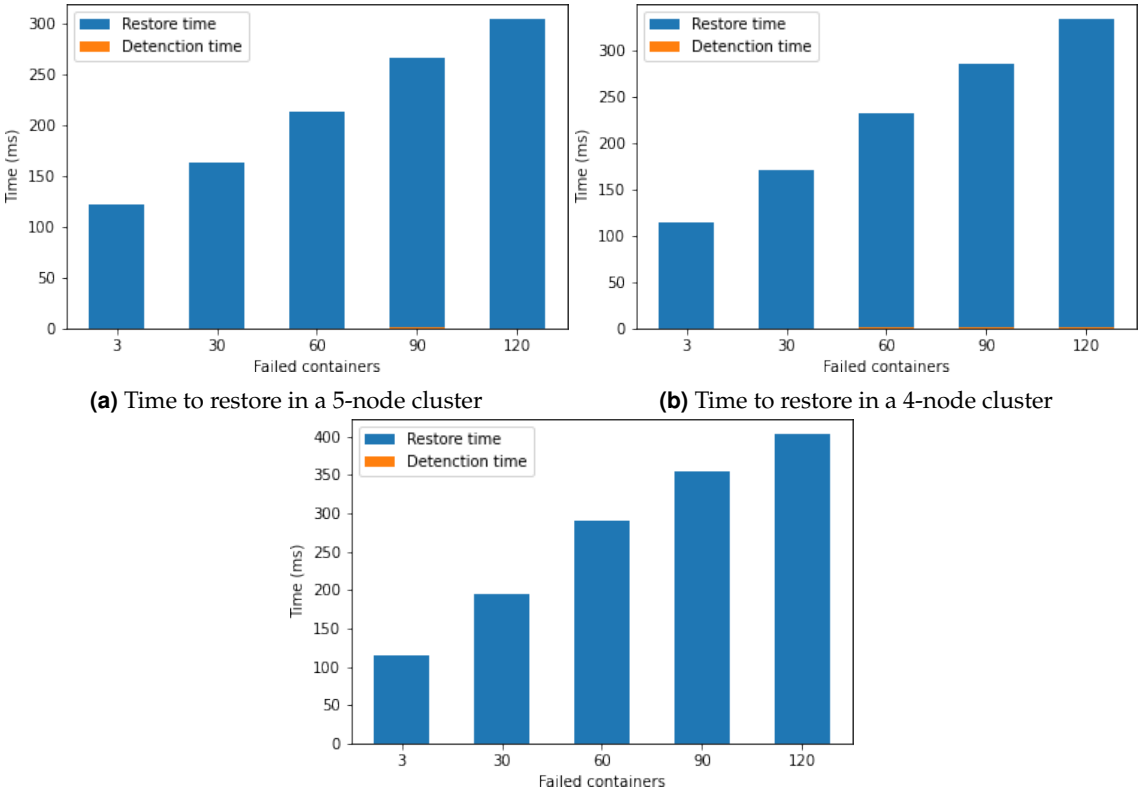
Table 7.1: Tolerancer Cluster's nodes characteristics.

7.4.2 Result discussion

In this section, we discuss the performance evaluation of Tolerancer from the service maintainability perspective.

Cluster size	Edge1	Edge2	Edge3	Edge4	Edge5
5 nodes	✓	✓	✓	✓	✓
3 nodes	✓	✓	✓	✗	✓
3 nodes	✓	✓	✗	✗	✓

Table 7.2: Tolerancer Clusters configurations



(a) Time to restore in a 5-node cluster

(b) Time to restore in a 4-node cluster

(c) Time to restore in a 3-node cluster

Figure 7.4: Time to restore in three different Tolerancer clusters

We set the timing to activate the interval value of the LOGGER equal to 5 seconds. Time selection is crucial in meeting the objective of the approach's design. If the interval period is too short, it overloads the system performance by performing more actions (*e.g.*, migration), and if it is too long, the approach may not immediately respond to the faults (or to the possible faults).

After deploying and running the services, we switched off one node in the three different clusters. We noticed that all Tolerancer 's components responded efficiently to such fault. A node failure leads to the following actions:

- The LOGGER of the failed node stops writing the health status information of the failed node,
- The ANALYZERs of the healthy nodes notice no information from the failed node, so

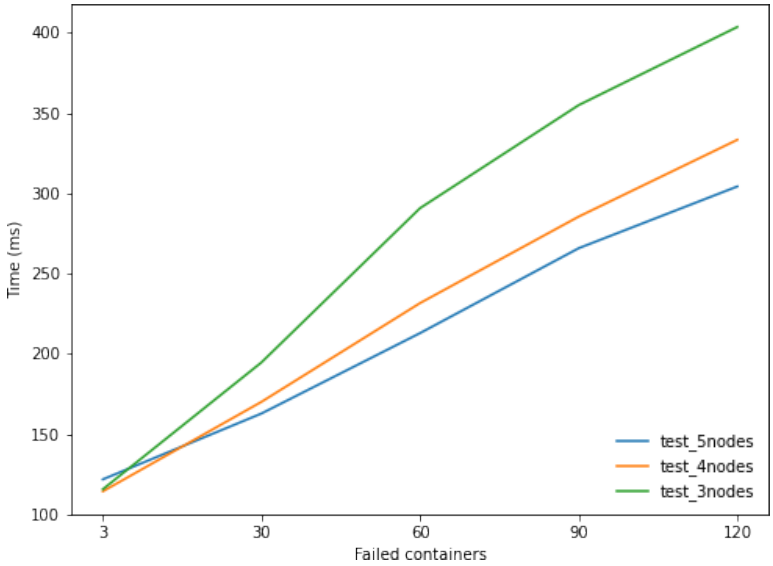


Figure 7.5: Comparing the time to restore in three different Tolerancer clusters

they try to contact it. After the no-response, the healthy nodes mark the status of the off node as failed.

- After detecting the failed node, the ANALYZERs of the healthy nodes get a list of the containers hosted in the failed node from the SHARED-MEMORY.
- The SCHEDULERs of the healthy nodes are triggered by the ANALYZERs, and the fastest of them reschedules the failed container to be hosted on a new node.
- The node that runs the scheduling process triggers the MIGRATOR of the destination node to accept the incoming container.
- The MIGRATOR of the destination node accepts the incoming container, runs it and updates the SHARED-MEMORY.

However, as most industrial applications are real-time applications, in each experiment, we measured the time needed to rerun the containers after any system failure. In Figure 7.4, the blue bars represent the time needed to restore the failed containers in the cluster with a fixed number of nodes. We can see that the time needed to restore the containers is directly proportional to the number of containers. This is expected as the migration requests are queued among all the remaining healthy nodes and hosted by the destination node(s) individually. The time required to restore any container is calculated as in Equation 7.4.1:

$$T_{Restore} = T_{Detect} + T_{Fcontainers} + \frac{T_{Rerun}}{N_{Hdevices}} \tag{7.4.1}$$

where:

- $T_{Restore}$ is the time required to restore the failed container,
- T_{Detect} is the time required to detect the failure,
- $T_{Fcontainer}$ is the time required to find the failed container,
- T_{Rerun} is the time required to rerun all the failed containers,
- $N_{Hdevices}$ is the number of healthy devices.

In the same figure, we can see the time needed to detect the failure, which is calculated as in Equation 7.4.2:

$$T_{Detect} = DetectFailure_{TS} - Failure_{TS} \tag{7.4.2}$$

where:

- $DetectFailure_{TS}$ is the detected failure timestamp.
- $Failure_{TS}$ is the failure timestamp.

It is clear from the figure that the failure detection time (which is represented by orange color) is quite less than the actual time needed to restore the services. The detection time is a small part of the total time needed to restore service. In other words, the figure shows that the longest period to restore the container is consumed by the scheduling and migration operations of the PlaU, not by the logging and analysis operations in the MonU.

To understand the effects of the cluster size (*i.e.*, the number of nodes in the cluster) on the performance, refer to Figure 7.5. It compares the time required to restore the containers in the three clusters presented in Figure 7.4. Each line describes a single cluster behavior. If we ignore the network failures, we can conclude that the larger cluster performs better, as the time needed to restore the services is less. The last part of Equation 7.4.1, $\frac{T_{Rerun}}{N_{Hdevices}}$, props this conclusion, as increasing the number of healthy nodes leads to a decrease in the value of this part, and consequently, decrease $T_{Restore}$.

Moreover, the figure also shows that as the number of failed containers increases, the gap between the clusters increases. This is because increasing the number of migration requests creates system congestion that leads to performance degradation.

7.5 Conclusion

The Chapter presents a continuum micro orchestrator, called Tolerancer , to solve the software and hardware-related failures and overloading in cloud edge-constrained environments. Tolerancer makes decisions to avoid or solve potential system faults. Experiments on a real testbed show that the proposed approach provides on-the-fly automatic actions to handle hardware and software-based failures. This version of the chapter includes the performance evaluation of the reactive part of Tolerancer , while the performance evaluation of the proactive part will be shown in an extended version.

We are currently working on expanding our approach to include more results by testing larger clusters and trying different time intervals to find its effects on the system. More constraint(s) could be considered (*e.g.*, deadline of the running services). Besides, the nodes at the cloud layer could be integrated with the nodes at the edge. In addition, managing other failure types (such as security-related failures) is a future direction of this work.

CHAPTER 8

Use Cases of Computing at the Continuun

This chapter addresses possible use cases and scenarios where computing at the Continuum becomes critical to solving old and new challenges in everyday life.

In section 8.1 We aim to use this OpenWolf to spread one of the most common workflows for a Continuum typical scenario. In particular, we will consider a smart city security camera system used for the image recognition of dangerous events as the scenario and a five-step deep learning workflow to collect, retrain, infer, and show any notable event. This workflow will be tested in different deployment configurations, and results will be discussed.

In Section 8.2, we focus on a Horizon Europe-funded project called TEMA, aims at addressing Natural Disaster Management through the use of sophisticated Cloud-Edge Continuum infrastructures by means of data analysis algorithms wrapped in Serverless functions deployed on a distributed infrastructure according to a Federated Learning scheduler that constantly monitors the infrastructure in search of the best way to satisfy required QoS constraints. In this section, we discuss the advantages of Serverless workflow and how they can be used and monitored to natively trigger complex algorithm pipelines in the continuum natively, dynamically placing and relocating them, taking into account incoming IoT data, QoS constraints, and the current status of the continuum infrastructure. Therefore we presented the Urgent Function Enabler (UFE) platform, a fully distributed architecture able to define, spread, and manage FaaS functions using local IOT data managed using the Fiware ecosystem and a computing infrastructure composed of mobile and stable nodes.

8.1 OpenWolf: Serverless Workflow Engine for AI on Continuum

This thesis work aimed to solve many problems related to Continuum Computing, proposing innovative solutions for deploying and orchestrating software, keeping the infrastructure secure, and responding to dynamic security constraints. To do that, we mainly used the Osmotic Computing paradigm and, most of all OpenWolf, a tool designed by us to build serverless workflows *Continuum native*. At this point, we have collected enough elements to test our platform and research in real use cases and scenarios that can be tailored to the continuum environment. Indeed, we are going to analyze a Deep Learning application for image classification, typical in a Smart City scenario. We will consider five steps: (i) collection, (ii) transformation, (iii) training, (iv) inference, and (v) plotting. Then, we will design an OpenWolf's manifest for this workflow and compare the performances deploying this both in a full-edge, full-cloud, Continuum environment.

As we have seen in Chapter 5, this scenario is widely used for many reasons, like the security surveillance of a public road or place or to monitor some environment parameters used to foresee weather anomalous conditions.

8.1.1 Smart City Use Case

Smart Cities are a typical scenario for the Continuum use case. In fact, it is easy to find the three Continuum layers (cloud, fog, and edge) over them. For instance, we could find IoT sensors and small computing devices in private and public spaces, like cameras and Raspberry Pi, for monitoring buildings, traffic, or environmental parameters. These data are then typically processed in local data factories provided by private citizens, municipalities, or research institutes, and often they trust private cloud providers like AWS or Azure, i.e., for long-time storage or processing. This chapter will analyze a typical pipe for image processing. Smart Cities rely on this algorithm to detect violent and dangerous situations, traffic rule violations, or roadside surveillance applications.

In the following, we will test an image processing workflow composed of five states. Each state represents a function, that is processed inside the workflow. Each state will be deployed inside a Continuum tier according to the static scheduling rule defined in the Workflow Manifest. According to the states' descriptions:

Collect: exploits a camera stream for collecting environment images.

Transform: edits the images, cleaning and filtering noisy data. It can be run on any of the

Continuum's tiers.

Train: trains a Recurrent Neural Network (RNN) model used to analyze the collected images.

Inference: predicts the input image's label using the latest model produced by the Train state.

Show: pushes the result of the inference over a web page.

The first problem we identify on the continuum, mostly when FaaS is implemented, is having a good scheduler for deploying functions according to specific QoS (i.e., latency, network bandwidth usage, resource performances). The second problem that relies on the first one is where to put data. These typically are collected on the Edge, but they could be partially computed on Edge or delivered to Cloud for massive analysis. QoS directly depends on the service we provide in the Smart City. For example, road traffic monitoring could require optimizing accuracy, whereas shotgun detection could require real-time analysis. Our proposed solution aims to give the possibility to directly customize what and where data are processed, trying to satisfy any kind of QoS, as we will see follow.

8.1.2 Design a Workflow using OpenWolf

Figure 8.1 shows the ideal and even the most common configuration for this kind of AI workflow. Basically, the edge layer is used to collect and infer data, while the fog is in charge of transforming and cleaning the data. Cloud tier is mostly demanded to train the inference model and show the inference's results. This workflow can be easily mapped in the Manifest format required by OpenWolf, the result is shown in the listing 8.1.

```
1  name: ML Workflow
2  callbackUrl: "http://.."
3  states:
4    measure:
5      function:
6        ref: fn-measure
7      start: true
8    transform:
9      function:
10       ref: fn-transform
```

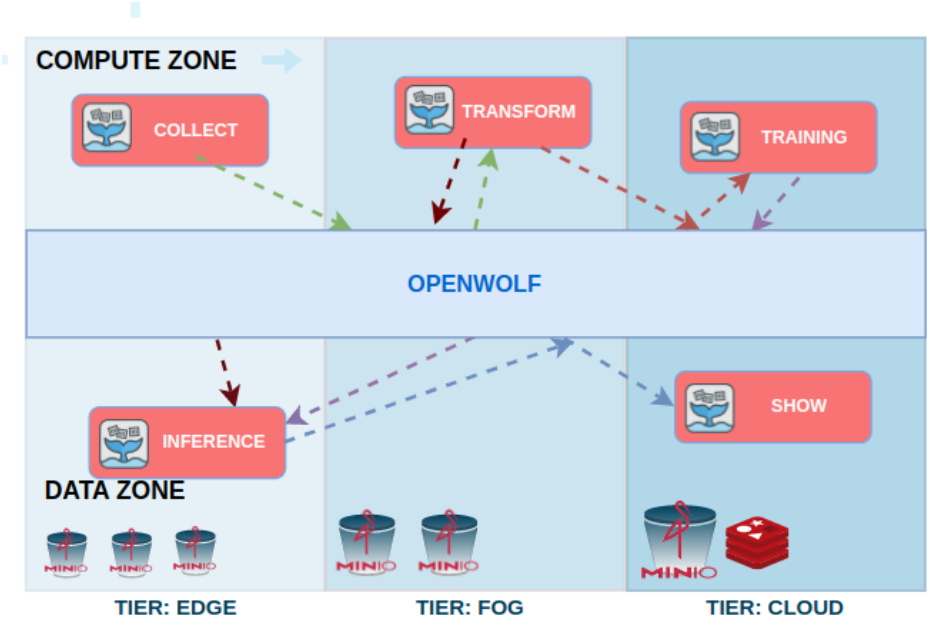


Figure 8.1: AI Workflow on OpenWolf

```
  train:
    function:
      ref: fn-train
  inference:
    function:  fn-predict
    end: true
   show:
    function:
      ref: fn-plot

functions:
  measure:
    endpoint: "https://gw/ \
      async-function/measure"
    config:
      resolution: medium
     [...]
workflow:
  measure:
    activation: True
  train:
    activation: measure
  inference:
    activation: measure
```

```
35  show:
36    activation: inference
```

Listing 8.1: Workflow AI Manifest

8.1.3 Performances

We evaluated the workflow in three different environments. We considered three key workflows' moments: (i) training, (ii) data fetching, and (iii) data inference both in a full cloud, full edge, and a continuum test bed. In the latter case, Cloud nodes were in charge of the model training, whereas Edge nodes were focused on collecting and inferencing data. These three functions have been encapsulated inside three different Openfaas functions. Based on the well-known CIFAR-10 dataset, the algorithms we used were based on a 50 epochs PyTorch training of a Recurrent Neural Network (RNN). The data training size is 130 MB, instead, the data test size, used during the inference, is around 100 MB. As an Edge node, we used a single Raspberry Pi 4, with an ARM64 operating system, 4 GB of RAM, and a 1.5 GHz quad-core processor. As a cloud node, we used a virtual machine with 16 GB of RAM, a 2.8 GHz quad-core processor, and an x64 operating system. Results are shown in Figure 8.2. As we see and expect, the Edge node needed 400% of the Cloud performance for training, and 120% for inferencing data, but the time to use local data is close to zero. On the other hand, Cloud requires 45 seconds to transfer data from the Edge Object Storage to the local storage. Moreover, the edge device does not require network usage, instead, Cloud will use the WAN network for receiving the entire test dataset from the Edge Object Storage. Finally, distributing the computation over the continuum environment allows the exploitation of the Cloud training time and the Edge data locality, avoiding any massive network usage. Unfortunately, as a direct consequence, the inference is done inside the Edge, but as shown in Figure, the overall performance in the continuum is better than both the Edge and the Cloud.

8.1.4 Conclusion

In this brief Chapter, we used OpenWolf to deal with a typical continuum workflow in a common scenario. We tested the platform considering a Smart City use case. In particular, we built a classical Deep Learning workflow, encapsulated each workflow's process inside a function, and deployed them among a continuum environment composed of an Edge node

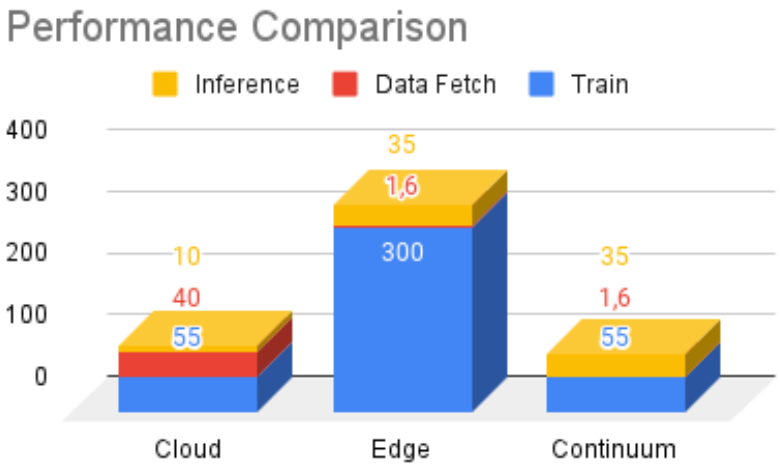


Figure 8.2: Workflow AI performance comparison on Continuum

and a Cloud node. Using the OpenWolf features, we tested the workflow performances for executing all the processes, and we compared the results considering a full-cloud, full-edge, and optimized continuum environment. This simple example, allowed us to demonstrate the validity of our solution to accomplish the continuum's needs and use cases, indeed with the good results obtained here, we can now explore in deep and in bigger contexts the application of OpenWolf as a production solution to making the Cloud-Edge real continuum environments.

8.2 TEMA: Event-Driven Serverless Workflows Platform for Natural Disaster Management

In recent years we have assisted in the rise of the Internet Of Things (IoT) as one of the fastest-growing data sources. The IoT has been applied in many different fields, like smart cities industries (Industry 4.0) and especially in environmental monitoring, with the purpose of keeping under control different parameters, such as noise, air quality or in more sophisticated infrastructures, the rise of some environmental disasters like fires, earthquakes or floods (Natural Disaster Management, **NDM**).

Independently by the application use case, IoT infrastructures are just data collectors and are not in charge of permanently storing, analyzing, and reacting to the data. When those latter actions are critical or time-sensitive, we fall into the area of Urgent Computing, which refers to using high-performance computing (HPC) resources to address critical and time-sensitive problems requiring rapid response. These problems include urgent scientific

research, emergency response planning, and real-time high-pressure decision-making.

The way to build HPC systems changed over the years, moving from adopting high-performance cloud infrastructures towards using distributed cloud edge cooperative infrastructures called Cloud-Edge Continuum or just *Continuum*.

The characteristics of a continuum infrastructure perfectly fit the needs of many time-critical scenarios, but, some challenging issues are still open, especially when applied to dynamic use cases like environment monitoring.

The TEMA project is a Horizon Europe (HE) project that addresses NDM needs by developing automated means for precise semantic area mapping and phenomenon evolution predictions for NDM in (near-)real-time. Potential end-users are mainly Civil Protection Agencies (CPAs), but also First Responders (FRs). To address this problem, TEMA proposes to design and develop an efficient continuum platform able to: (i) Increase responsiveness/speed of extreme data analysis algorithms; (ii) optimize the computation, dynamically migrating it accordingly with the just collected data, the historical information and the Quality of Service (QoS) needed; (iii) drive the computation considering the events that are measured by the IoT infrastructure.

In this work, we want to present the preliminary work inside the TEMA project, proposing a continuum native, event-driven workflow architecture based on the Function as a Service (FaaS) paradigm. The goals of this architecture are the following:

- spreading functions at any continuum tier, letting the system use the best available infrastructure to run an NDM workflow;
- monitoring and controlling the FaaS infrastructures in order to build an up-to-date dataset from which to learn where to compute based on the expected QoS and the forecast one;
- dynamically connecting functions on the continuum to trigger complex distributed workflows able to analyze an incoming event properly.

8.2.1 State of the Art

Since IoT has risen, many different use cases have appeared with the aim of measuring and preventing some possible disasters [147]. In some industries, IoT can be used to extract oil and gas, a well-known system prone to incidents.[148]; while in critical areas, IoT is used to detect possible flooding and earthquakes, eventually triggering local alarms [149, 150]; or

also to detect tsunamis [151] in oceanic coast areas. IoT is often associated with implementing Machine Learning algorithms used to analyze those data, extract knowledge, and then forecast something about the next possible events[152, 153, 154].

Unfortunately, working on time-sensitive use cases using machine learning algorithms can often be tricky due to algorithm heaviness, or distance from the data location [155, 156].

Urgent computing was born to provide the best resources possible as soon as needed to absolve time-critical events like environmental disasters or human health monitoring [157]. In literature, there exist many directions adopted to realize urgent computing. Cloud Computing, of course, thanks to its flexibility in providing any kind of resource has been immediately adopted to realize HPC system to accomplish a job sooner [158, 159], but as often highlighted, even less powerful system, but closer to the data can better accomplish a time critical job [160]. The advantages of the use of Edge Computing in fact are mostly related to the proximity to the data [161] and the possibility to work even in the presence of partial network partitions [5], thanks to these advantages some authors proposed Edge as a valid infrastructure to run time-critical analysis [162, 163].

Continuum Computing has been extensively used in those scenarios. In [50], the authors propose an edge framework, adapted for the continuum, to spread the computation across the continuum using user-defined dynamic rules that can be defined even taking into account the urgency of the computation.

In [66] the authors collect a list of jobs to be scheduled, taking into account the emergency of each of them, then apply a federated learning algorithm to assign them to a specific node in the continuum federation, using the previous data sets (Qos Required, Qos reached, node, task) as a source of data from which learn. Using machine learning to optimize the QoS of a given task has been further used [67], but as even the authors have highlighted, knowing the nature of the task and then forecasting its behavior is not easy, and prediction risks being wrong.

With the advent of containerization, researchers have seen in orchestrators, especially Kubernetes a great tool to federate heterogeneous environments like the continuum, and then customize it to deploy containerized applications on a node that can satisfy a time-based QoS constraint [164]; taking in example [65], authors use Kubernetes to federate the continuum, then they provide an internal opensource custom scheduler to deploy pods on a node that can satisfy network latency constraints, a factor that became crucial in real-time processing.

All the previous works have the assumption that any containerized application can be run *anywhere* and that it is possible to profile it, to forecast its behavior, and both considerations

are not strictly true, until the introduction of Serverless Computing and FaaS, that we have already extensively discussed.

8.2.2 Architecture

In this section, we present the theoretical architecture we aim to include inside the TEMA project to enable Urgent Computing at the Continuum in the NDM context. This platform shorted by the name Urgent Function Enabler (UFE) is a distributed architecture composed of six main units: 1. the Infrastructure unit (IIA); 2. the IoT unit (IA); 3. the computation unit (CU); 4. the monitoring unit (MU); 5. the scheduler unit (SU); 6. the workflow unit (WU);

Infrastructure unit

The infrastructure unit is strictly related to the CU, and is basically made up of all the nodes that are able to deal in any way with the data. In turn, we distinguish two main roles in the IIU which are the sensors and the workers nodes. The sensors are at the lowest level, and they are basically able just to measure environmental parameters and provide them as system data.

The workers do not measure data, but they are able to receive them and then apply some kind of computation like a transformation, or data analysis, Eventually the result of that can be reintegrated into the system in order to be collected and analyzed by more workers.

The workers can belong to two main classes: 1. static workers; 2. mobile workers.

Static workers compose the most traditional computing infrastructure that basically is composed of cloud nodes deployed on remote data centers; edge nodes close to the data sources, realized using Raspberrys, Intel Nuc devices, or other micro computers and fog nodes, usually implemented with workstations or small servers racks distributed along the path from the edge to the cloud.

Mobile workers, it is a kind of novelty in this field. We consider mobile any device that can change physically change its position, we include in this group devices such as robots and drones.

Drones and robots have already been used in the field of NDM; some real examples are in [165] where drones were used to monitor wildfires in California. The drones were able to fly over the fires and collect data on their size and intensity. This information was then used to help firefighters make decisions about how to fight the fires; and in [166] drones were used to track poachers in Africa. The drones were able to fly over the camps of poachers and identify

them. The information was then used by law enforcement to arrest the poachers.

Mobile workers, as well as static ones, can be used by the other units to run algorithms based on the data they can easily reach; to do that, the computation unit has to be used.

IoT Unit

The IA is the lowest infrastructure in UFE, it consists of all sensors used to monitor the environment, and it is managed by a Fiware infrastructure that allows the device to be authenticated and authorized but also provides API to send data and receive commands from and to the upper level [5]. The main components of this architecture are the Orion Context Broker, the IoT Agent (IOTA), the P2P-IDM[5], and the PEP Proxy.

- The P2P IDM is a distributed eventually consistent unit that stores all service accounts used by IOT devices; it provides the API to obtain and verify Oauth2 tokens. The advantage of using a P2P IDM with respect to a centralized IDM is the possibility of keeping the service up in the presence of network partitions or disconnections, and this is crucial when the infrastructure is composed of edge and mobile nodes.

- The IOTA is a middleware that receives raw data from the IoT, transforms it in NGSI format and then sends them to the Orion Context Broker.

- The PEP proxy is an authorization proxy, placed in front of the IOTA, that verifies the device authorization according to the Keyrock policies and then forwards or denies the request to the IOTA.

- The Orion Context Broker is the core of the IOT Unit. It receives the IoT data from the IOTA NGSI format, and then offers them to third clients using an advanced Pub/Sub model; the subscription will be used to trigger faas workflows on the continuum;

The Computation Unit

The CU is responsible for executing the algorithms designed to compute the data collected from the IoT unit. All algorithms are encapsulated in functions since this unit is strictly based on the FaaS paradigm. In brief, FaaS is able to encapsulate a stateless function inside the container, letting external clients invoke this function using typically HTTP or pub/sub-patterns.

In UFE, the CU is not a unique infrastructure, rather it is the logical union of all the CU installed in all the nodes that compose the continuum infrastructure.

Finally, the main component of the CU is OpenFaas. OpenFaas is the most stared open-source engine on GitHub [1], it is composed of a gateway that lets us invoke any function using HTTP; a NATS server associated with a Queue-Worker, which lets us invoke the function using pub/sub model, sending back the result using webhooks, and a faas-cli, which is used to interact with OpenFaas, building multiarchitecture function natively, and of course, deploying them. OpenFaas is a centralized Function Registry to store the functions available in all the CU; this registry is even cached locally to avoid the cold start problem, typical in any containerized infrastructure.

As anticipated previously, the CU works integrated with the IIU; in particular, the FaaS stack is supposed to run in the workers' nodes, in fact, the massive use of low-level containerization together with a light system such as OpenFaaS, makes it possible to run functions in most of the traditional mobile units as well as a static unit we want to integrate. At the end then, we will be able to run algorithms on a drone, a robot as well as in a cloud virtual machine.

The Monitoring Unit

The MU is fundamental for planning a time-critical computation. The MU is installed along with the computation unit and collects all the useful metrics of all the functions that are executed in the CU. The Monitoring Unit is made up of a high-performance proxy (HPC), a Prometheus instance, and a centralized shared data lake. The HPC is posed in front of the OpenFaas gateway, then it intercepts all the function invocations, forwarding the result to the client, but before doing that it retrieves from the OpenFaas response the request ID that is used asynchronously to fetch all the information from Prometheus and OpenFaas' logger. Prometheus is a popular open-source monitoring and alerting system that is used to collect and analyze metrics from various sources in real-time, but at this moment the information that the MU fetches are the following: 1. function executed; 2. continuum's node where the function is run; 3. start time and duration of the computation; 4. start and end time of the request; 5. CPU cycles and memory used to run that function and conclude the request. All these metrics are collected together and then sent to the Data Lake component, which will permanently store them, to be used later to apply scheduler choices.

[1] https://github.com/openfaas/faas

The Scheduler Unit

A key component in the described architecture is related to the scheduling of created workflows among the components that belong to the Cloud-Continuum Infrastructure. In particular, the SU exploits context data to estimate the more efficient and convenient node in which the function should be deployed. A key role is played by *MU* that collects the monitored data that will actually be used to guide the scheduler in the offloading of workflow processes. The scheduling is dynamic, and it exploits Machine Learning inference to establish more appropriate nodes in which workflow can be deployed. The inference of a Machine Learning model could be considered a complex operation that can compromise a time-constrained application. For this reason, the scheduler will figure out an inference with a pre-trained model in an asynchronous fashion when a new workflow is not yet created or deployed. The model consists of a Deep Neural Network in which the inputs are all metrics collected through *MU* and the QoS score brought by the specific workflow. The model output consists of a score that ranks each node in the architecture on the basis of different parameters: time-response with respect to the scheduler, used memory, used CPU, and other possible parameters collected by Prometheus instance present in *MU*. In particular, it performs, exploiting a classical *Soft Max Activation Function*, a classification of more appropriate nodes according to computed scores. It classifies the most efficient node candidates for deploying the next workflow processes. The decision of the nodes for each process is decided following the probabilities computed by the model. Moreover, the model will be updated by exploiting historical data collected through a continuous learning mechanism that takes advantage of the Federated Learning approach [167]. Indeed, each node, periodically, trains a local model that will be aggregated in the central cloud server and exploited by the scheduler's central component. The scheduler will become more precise thanks to historical data collected by MU.

The Workflow Unit

One of the most highlighted downsides of FaaS is the inability to deploy complex and distributed workflows that connect functions to serve a bigger and composed computation. To overcome this, most of the FaaS-based architectures propose to use a centralized microservice that invokes and synchronizes the right function when needed. This small trick might be not the best choice for any distributed time-critical environments; therefore, we integrated the WU. The WU let us define serverless workflows using a light version of OpenWolf

[2]. OpenWolf is a small distributed broker that can be instantiated on Kubernetes or with just Docker for the lightest environments in this use case, we have an OpenWolf agent for each computing tier. OpenWolf [2] uses a customized version of the Serverless Workflow DSL [3] to describe how the input and output functions are connected, allowing one to define asynchronous faas DAGs. Using the DSL, we can submit a *Manifest file* to the OpenWolf agent, The manifest file will be used to declare a workflow and then it will be used to trigger the function defined inside it when a specific event occurs. Through the Manifest is even possible to decide where to deploy a function to use inside the workflow, but in this case, we modified the function allocation, in order to migrate a function location according to the QoS of an event and the suggestion coming from the SU. Manifests are even more shared with all the OpenWolf agents, in this way, any instance can run the same workflow.

Units Integration

The integration of the UFE unit is quite straightforward. The overall architecture is presented in figure 8.3, from the bottom we distinguished three different zones provided with the IoT sensors. Each device can use any of D-IDM to get an Oauth2 token, that is used to access the IOT Agent in the IA. In the same unit, the IoT Agent sends the data to the Context Broker, where the data live. In parallel, as soon as a Workflow manifest is sent to OpenWolf, the functions included in that file are deployed on the Computing Infrastructure according to the scheduler's choices. When all the functions are ready, the workflow endpoint is subscribed to the Context Broker which contains the data needed to trigger the workflow. When the subscription arrives at the Context Broker, data will be sent to the first function on the workflow, and in turn, the result will be forwarded to the other functions until all the functions in the workflow are not consumed. While the functions are run, the monitor system installed alongside the serverless platform records all the information related to the executions. This information will travel to the Data Lake, which the scheduler will continuously read to adapt the function position in the infrastructure dynamically.

8.2.3 Conclusion

The work proposes a solution for Natural Disaster Management in the Horizon TEMA project exploiting Cloud-Continuum workflows, and applying them considering QoS parameters. TEMA is a Horizon Europe-funded project that wants to manage Natural disaster

[2]https://github.com/christiansicari/OpenWolf
[3]https://serverlessworkflow.io/

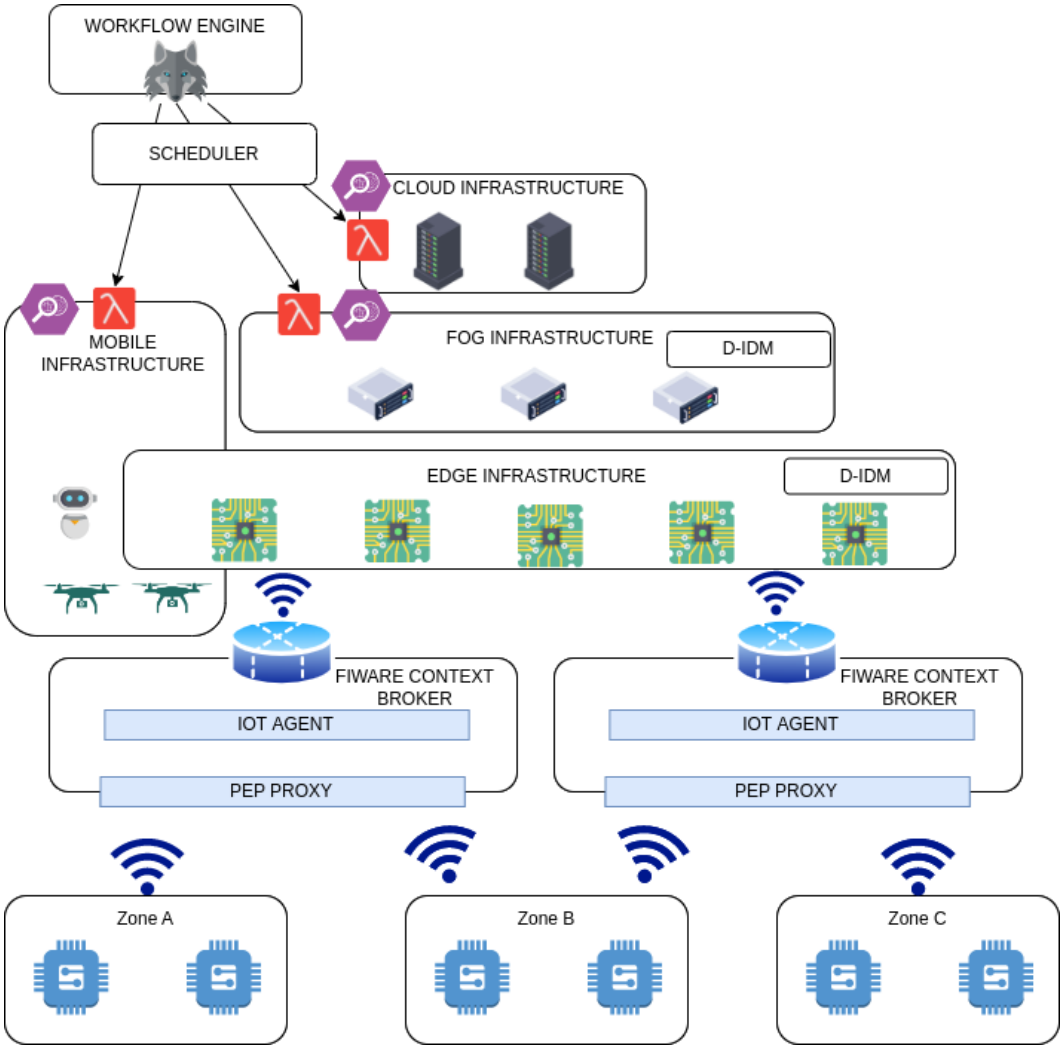


Figure 8.3: TEMA platform architecture

Management exploiting a Cloud-Edge Continuum infrastructure. For this reason, the architecture depicted in this work must be able to manage Urgent Computation applications.

The solution described here is a concrete workflow system use case that can satisfy the QoS and realize a complex algorithm pipeline exploiting IoT-collected data in a Cloud-Continuum infrastructure. Moreover, the architecture described is capable of performing dynamic scheduling of workflow components, exploiting the monitored data by each node of infrastructure, and a sophisticated machine learning model capable of retrieving the optimal nodes. The fixed steps to perform are the practical application of the solution in a concrete use case provided by the TEMA project and the practical performance evaluation of the implemented architecture. As naturally expected in future works, we need to validate the architecture we proposed, in order to understand the limits and strengths of the works.

CHAPTER 9

Conclusion and Future Works

In this thesis work, we aimed to address many issues that disallow making computing on Continuum possible, and we tried to solve all of them by applying many innovative solutions. Taking into account the most updated State of the Art in the fields of Cloud and Edge Computing, Continuum, and in general, Distributed Computing, we identified six issues that make computing at the Continuum hard to implement: (i) integrating public and private clouds, federating them, and deploying "pieces" of applications independently; (ii) deploying continuum native applications to make the most of the environment where they execute; (iii) guaranteeing security during the interactions across the infrastructure; (iv); discovering and locating applications that can move across the continuum environment; (v) balancing and keeping reliable a continuum infrastructure.

In order to address all these issues, we first analyzed the background, in particular, in Chapter 2 we analyzed the reason why Continuum arose and what technologies are behind it, in particular Cloud and Edge Computing. Then we analyzed new patterns to deploy applications, particularly the microservice architecture and the serverless, finally, we discussed Osmotic Computing, a high-level paradigm that describes tools, approaches, and concepts to enable the computation at the continuum.

After this background analysis, we came back to the highlighted issues, and we tried to solve all of them, in order, using every time what we did in the previous step. Therefore, in Chapter 3, we proposed a standard to deploy applications on the Continuum, by means of this standard we aimed to make possible the federation between private and public cloud

and edge infrastructures, and even to propose a blueprint about how to deploy continuum native applications.

In Chapter 4, we used the earned knowledge from the previous work to implement OpenWolf and to improve RPulsar. OpenWolf is a serverless workflow engine able to spread and connect serverless functions transparently across any continuum infrastructure. By means of OpenWolf, we found a way to deploy continuum native applications, therefore solving the second highlighted issue. while, improving RPulsar we gave the possibility to dynamically combine data producers and consumers on demand, matching and connecting them spreading functions according to the provided matching rules. In Chapter 5 we dug into many security issues that can compromise computing at the continuum. In these chapters, we widely used the concepts inherited from Osmotic Computing to guarantee addressability, accountability, integrity and confidentiality for the data exchanged across the continuum environment, by means of the definition of osmotic membranes, decentralized peer-to-peer and VPN-based overlay networks.

After we have found a way to deploy standardized and secure applications on the Continuum, in Chapter 6, we developed the OCE-DNS, a geohash-based DNS, used both to discover services on Continuum but also to register them, hiding possible migrations and down times. Inside this Chapter, we have even introduced the Extended Plus Code, a three-dimensional hierarchical hash code, that we used to address services, but it has been used even in other works [168], to map virtual and real reality.

As highlighted in the background of this thesis, but also in the chapters, orchestrating applications in the continuum is not easy, and most used orchestrators like Kubernetes or Nomad might fail due to the diversity of the environment they have to deal with or just because the constraints given by the capacity of some nodes inside the infrastructure. Starting from these considerations we developed Tolerancer, described in Chapter 7. Tolerancer is a peer-to-peer micro orchestrator that monitors a container-based continuum environment, restoring failed applications and avoiding system overloading by properly migrating the containers across the infrastructure's nodes.

Finally, we tested all these works in a smart city scenario, indeed in Chapter 8 we performed a performance analysis of a deep learning workflow using OpenWolf as a baseline and a continuum as infrastructure. In this work, we demonstrated how is possible to deploy continuum native applications, and how is possible to exploit the characteristics of each layer in the continuum to get the best result possible to achieve an objective. Following we discussed the TEMA project, a European-founded project that is going to take advantage of

our Continuum solution to address urgent computing scenarios.

Continuum Computing is probably one of the hottest topics in the computer science research field, and we aim to further advance the state of the art in this area, working on more use cases and integration in the future. In this regard, we would embrace a broader concept such as SysOps, which aims at unifying the concept of maintainability and operability of a platform. Our idea in a nutshell is to provide a workflow system that prepares a Continuum environment as well to optimize the workflow to take advantage of the environment itself. Currently, we are working to improve the scheduling of the computation on the continuum, making it smart, which means choosing where to compute something using the history, the application constraints, and the current system status as decision factors. We are even working on the dynamic composition of applications based on the match of compatibility profile. This idea inspired, by the state of the art can be integrated with OpenWolf and in general with all the previous work, to respond better to the dynamicity of the Continuum environment. All these updates, once published will be even released as an open-source project, to let people join and improve our work and research.

Bibliography

[1] Christian Sicari, Alessio Catalfamo, Lorenzo Carnevale, Antonino Galletta, Antonio Celesti, Maria Fazio, and Massimo Villari. *Toward the Edge-Cloud Continuum Through the Serverless Workflows*, pages 1–18. Springer Nature Switzerland, Cham, 2024. (Cited at page x)

[2] Christian Sicari, Lorenzo Carnevale, Antonino Galletta, and Massimo Villari. Openwolf: A serverless workflow engine for native cloud-edge continuum. In *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 1–8, 2022. (Cited at pages x, 57, 66 e 142)

[3] Christian Sicari, Daniel Balouek-Thomert, Manish Parashar, and Massimo Villari. Event-driven faas workflows for enabling iot data processing at the cloud edge continuum. In *Proceedings of the 16th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '23, Taormina, Messina, Italy, 2023. Association for Computing Machinery. under review in the 16th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC 2023). (Cited at page x)

[4] Gabriele Morabito, Christian Sicari, Armando Ruggeri, Antonio Celesti, and Lorenzo Carnevale. Secure-by-design serverless workflows on the edge–cloud continuum through the osmotic computing paradigm. *Internet of Things*, 22:100737, 2023. (Cited at pages x e 42)

[5] Christian Sicari, Alessio Catalfamo, Antonino Galletta, and Massimo Villari. A distributed peer to peer identity and access management for the osmotic computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 775–781, 2022. (Cited at pages xi, 59, 137 e 139)

[6] Christian Sicari, Antonino Galletta, Antonio Celesti, Maria Fazio, and Massimo Villari. An osmotic computing enabled domain naming system (oce-dns) for distributed service relocation between cloud and edge. *Computers & Electrical Engineering*, 96:1–25, 05 2021. (Cited at pages xi e 75)

[7] Auday Al-Dulaimy, Christian Sicari, Alessandro V. Papadopoulos, Antonino Galletta, Massimo Villari, and Mohammad Ashjaei. Tolerancer: A fault tolerance approach for cloud manufacturing environments. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2022. (Cited at page xi)

[8] Christian Sicari, Lorenzo Carnevale, Antonino Galletta, and Massimo Villari. Openwolf: Serverless workflow engine for ai on continuum. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, 09 2022. (Cited at page xi)

[9] Christian Sicari, Alessio Catalfamo, Lorenzo Carnevale, Antonino Galletta, Daniel Balouek-Thomert, Manish Parashar, and Massimo Villari. Tema: Event driven serverless workflows platform for natural disaster management. In *2023 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2023. (Cited at page xi)

[10] Schahram Dustdar, Victor Casamajor Pujol, and Praveen Kumar Donta. On distributed computing continuum systems. *IEEE Transactions on Knowledge and Data Engineering*, XX:1–14, 2022. (Cited at pages 2, 15 e 39)

[11] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic computing: A new paradigm for edge/cloud integration. *IEEE Cloud Computing*, 3(6):76–83, 2016. (Cited at pages 11, 57, 78 e 80)

[12] L. Carnevale, A. Celesti, A. Galletta, S. Dustdar, and M. Villari. From the cloud to edge and iot: a smart orchestration architecture for enabling osmotic computing. 2018. (Cited at page 11)

[13] M. Villari, M. Fazio, S. Dustdar, O. Rana, L. Chen, and R. Ranjan. Software defined membrane: Policy-driven edge and internet of things security. *IEEE Cloud Computing, Volume: 4*, 2017. (Cited at page 12)

[14] Christian Sicari, Antonino Galletta, Antonio Celesti, Maria Fazio, and Massimo Villari. An osmotic computing enabled domain naming system (oce-dns) for distributed service relocation between cloud and edge. *Computers & Electrical Engineering*, 96:107578, 2021. (Cited at page 12)

[15] A. Buzachis and M. Villari. Basic principles of osmotic computing: Secure and dependable microelements (mels) orchestration leveraging blockchain facilities. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018. (Cited at page 13)

[16] Harald Mueller, Spyridon V. Gogouvitis, Houssam Haitof, Andreas Seitz, and Bernd Bruegge. Poster abstract: Continuous computing from cloud to edge. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 97–98, 2016. (Cited at page 14)

[17] Anurag Ranjan, Francesc Guim, Mandar Chincholkar, Prakash Ramchandran, Ranjan Mishra, and Sunku Ranganath. Convergence of edge services & edge infrastructure. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 96–99, 2021. (Cited at pages 14 e 15)

[18] A. Luckow, K. Rattan, and S. Jha. Pilot-edge: Distributed resource management along the edge-to-cloud continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 874–878, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society. (Cited at pages 15, 41 e 60)

[19] Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10, 2019. (Cited at page 15)

[20] Tobias Pfandzelter and David Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24, 2020. (Cited at page 15)

[21] Michele Ciavotta, Davide Motterlini, Marco Savi, and Alessandro Tundo. Dfaas: Decentralized function-as-a-service for federated edge computing. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2021. (Cited at page 15)

[22] Xavi Masip-bruin, Eva Marín-tordera, Sergi Sánchez-lópez, Jordi Garcia, Admela Jukan, Ana Juan Ferrer, Anna Queralt, Antonio Salis, Andrea Bartoli, Matija Cankar, Cristovao

Cordeiro, Jens Jensen, and John Kennedy. Managing the cloud continuum: Lessons learnt from a real fog-to-cloud deployment. *Sensors*, 21(9), 2021. (Cited at page 15)

[23] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33:1159–1174, 11 2019. (Cited at page 15)

[24] Dragi Kimovski, Christian Bauer, Narges Mehran, and Radu Prodan. Big data pipeline scheduling and adaptation on the computing continuum. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1153–1158, 2022. (Cited at pages 15 e 39)

[25] Luiz Bittencourt, Roger Immich, Rizos Sakellariou, Nelson Fonseca, Edmundo Madeira, Marilia Curado, Leandro Villas, Luiz DaSilva, Craig Lee, and Omer Rana. The Internet of Things, Fog and Cloud continuum: Integration and challenges. *Internet of Things (Netherlands)*, 3-4:134–155, 09 2018. (Cited at page 15)

[26] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3, 10 2019. (Cited at page 15)

[27] Debnath Mukherjee, Debraj Pal, and Prateep Misra. Workflow for the internet of things. pages 745–751, 01 2017. (Cited at page 15)

[28] Raouf Boutaba. The cloud to things continuum. pages 7–7, 2021. (Cited at page 15)

[29] Xavier Merino, Carlos Otero, David Nieves-Acaron, and Benjamin Luchterhand. Towards orchestration in the cloud-fog continuum. *Conference Proceedings - IEEE SOUTHEASTCON*, 2021-March, 2021. (Cited at page 15)

[30] Ivan Cilic, Ivana Podnar Zarko, and Mario Kusek. Towards service orchestration for the cloud-to-thing continuum. *2021 6th International Conference on Smart and Sustainable Technologies, SpliTech 2021*, 2021. (Cited at pages 15 e 39)

[31] Nuno Faria, Daniel Costa, José Pereira, Ricardo Vilaça, Luís Ferreira, and Fábio Coelho. Aida-db: A data management architecture for the edge and cloud continuum. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6, 2022. (Cited at page 15)

[32] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 17–25, 2022. (Cited at page 15)

[33] Efterpi Paraskevoulakou and Dimosthenis Kyriazis. Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud continuum. *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN 2021*, pages 110–117, 2021. (Cited at pages 15 e 25)

[34] Alessandro Bocci, Stefano Forti, Gian-Luigi Ferrari, and Antonio Brogi. Type, pad, and place: Avoiding data leaks in cloud-iot faas orchestrations. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 798–805, 2022. (Cited at page 15)

[35] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 89–103, New York, NY, USA, 2017. Association for Computing Machinery. (Cited at pages 16 e 25)

[36] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18:1039–1065, 08 2006. (Cited at page 16)

[37] Marcin Płóciennik, Tomasz Zok, Antonio Gómez-Iglesias, Francisco Castejón, Andrés Bustos, Manuel Aurelio Rodríguez-Pascua, and José Luis Velasco. Workflows orchestration in distributed computing infrastructures. *Proceedings of the 2012 International Conference on High Performance Computing and Simulation, HPCS 2012*, pages 616–622, 2012. (Cited at page 16)

[38] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3165:11–20, 2004. (Cited at page 16)

[39] Kevin Lee, Norman W. Paton, Rizos Sakellariou, Ewa Deelman, Alvaro A.A. Fernandes, and Gaurang Mehta. Adaptive workflow processing and execution in pegasus. In *2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, pages 99–106, 2008. (Cited at page 16)

[40] Ewa Deelman, Karan Vahi, Mats Rynge, Gideon Juve, Rajiv Mayani, and Rafael Ferreira da Silva. Pegasus in the cloud: Science automation through workflow technologies. *IEEE Internet Computing*, 20(1):70–76, 2016. (Cited at page 16)

[41] Dragi Kimovski, Roland Mathá, Josef Hammer, Narges Mehran, Hermann Hellwagner, and Radu Prodan. Cloud, fog, or edge: Where to compute? *IEEE Internet Computing*, 25(4):30–36, 2021. (Cited at page 16)

[42] Karan Vahi, Mats Rynge, George Papadimitriou, Duncan A. Brown, Rajiv Mayani, Rafael Ferreira da Silva, Ewa Deelman, Anirban Mandal, Eric Lyons, and Michael Zink. Custom execution environments with containers in pegasus-enabled scientific workflows. In *2019 15th International Conference on eScience (eScience)*, pages 281–290, 2019. (Cited at page 16)

[43] Qingye Jiang, Young Choon Lee, and Albert Y. Zomaya. Serverless execution of scientific workflows. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10601 LNCS:706–721, 2017. (Cited at page 16)

[44] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018. (Cited at page 16)

[45] Wolfgang Gerlach, Wei Tang, Andreas Wilke, Dan Olson, and Folker Meyer. Container orchestration for scientific workflows. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, (March):377–378, 2015. (Cited at page 16)

[46] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems*, 110:502–514, 2020. (Cited at page 16)

[47] Tom Diethe, Meelis Kull, Niall Twomey, Kacper Sokol, Hao Song, Miquel Perelló-Nieto, Emma Tonkin, and Peter A. Flach. Hyperstream: a workflow engine for streaming data. *CoRR*, abs/1908.02858, 2019. (Cited at page 16)

[48] Andrzej Jasinski, Yuansong Qiao, John Keeney, Enda Fallon, and Ronan Flynn. A workflow engine server for the design of adaptive and scalable workflows. *30th Irish Signals and Systems Conference, ISSC 2019*, (June), 2019. (Cited at page 16)

[49] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: Trigger-based orchestration of serverless workflows. *DEBS 2020 - Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, (Debs):3–14, 2020. (Cited at page 16)

[50] Eduard Gibert Renart, Daniel Balouek-Thomert, and Manish Parashar. An edge-based framework for enabling data-driven pipelines for iot systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 885–894, 2019. (Cited at pages 17, 40, 48 e 137)

[51] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40, 2017. (Cited at page 17)

[52] Eduard Renart, Daniel Balouek-Thomert, Xuan Hu, Jie Gong, and Manish Parashar. Online decision-making using edge resources for content-driven stream processing. In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 384–392, 2017. (Cited at page 17)

[53] Zeina Houmani, Daniel Balouek-Thomert, Eddy Caron, and Manish Parashar. Enabling microservices management for deep learning applications across the edge-cloud continuum. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–146. IEEE, 2021. (Cited at pages 17 e 25)

[54] Openfaas Workflow, howpublished = `https://github.com/s8sg/faas-flow`, note = Last access May 2022. (Cited at page 26)

[55] Fission Workflow, howpublished = `https://github.com/fission/fission-workflows`, note = Last access May 2022. (Cited at page 26)

[56] Preeti Yadav and Sandeep Vishwakarma. Application of internet of things and big data towards a smart city. In *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, pages 1–5, 2018. (Cited at page 39)

[57] Jiachen Wang, Ji Ma, Kangping Hu, Zheng Zhou, Hui Zhang, Xiao Xie, and Yingcai Wu. Tac-trainer: A visual analytics system for iot-based racket sports training. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):951–961, 2023. (Cited at page 39)

[58] Parag Chatterjee, Leandro J. Cymberknop, and Ricardo L. Armentano. Iot-based decision support system for intelligent healthcare — applied to cardiovascular diseases. In *2017 7th International Conference on Communication Systems and Network Technologies (CSNT)*, pages 362–366, 2017. (Cited at page 39)

[59] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, 2018. (Cited at page 39)

[60] Jacopo Massa, Stefano Forti, and Antonio Brogi. Data-aware service placement in the cloud-iot continuum. In Johanna Barzen, Frank Leymann, and Schahram Dustdar, editors, *Service-Oriented Computing*, pages 139–158, Cham, 2022. Springer International Publishing. (Cited at page 39)

[61] Stefano Forti and Antonio Brogi. Green application placement in the cloud-iot continuum. In James Cheney and Simona Perri, editors, *Practical Aspects of Declarative Languages*, pages 208–217, Cham, 2022. Springer International Publishing. (Cited at page 39)

[62] Antero Taivalsaari, Tommi Mikkonen, and Cesare Pautasso. Towards seamless iot device-edge-cloud continuum:. In Maxim Bakaev, In-Young Ko, Michael Mrissa, Cesare Pautasso, and Abhishek Srivastava, editors, *ICWE 2021 Workshops*, pages 82–98, Cham, 2022. Springer International Publishing. (Cited at page 39)

[63] Victor Casamayor Pujol, Andrea Morichetta, Ilir Murturi, Praveen Kumar Donta, and Schahram Dustdar. Fundamental research challenges for distributed computing continuum systems. *Information*, 14(3), 2023. (Cited at page 39)

[64] Bin Cheng, Gürkan Solmaz, Flavio Cirillo, Ernö Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, 5(2):696–707, 2018. (Cited at pages 41 e 42)

[65] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174, 2020. (Cited at pages 41, 42 e 137)

[66] Gabriele Proietti Mattia and Roberto Beraldi. Leveraging reinforcement learning for online scheduling of real-time tasks in the edge/fog-to-cloud computing continuum. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pages 1–9, 2021. (Cited at pages 41 e 137)

[67] Albert Jonathan, Abhishek Chandra, and Jon Weissman. Awan: Locality-aware resource manager for geo-distributed data-intensive applications. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 32–41, 2016. (Cited at pages 41 e 137)

[68] Bin Cheng, Jonathan Fuerst, Gurkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive iot services. In *2019 IEEE International Conference on Services Computing (SCC)*, pages 28–35, 2019. (Cited at page 42)

[69] Nicolas Ferry, Rustem Dautov, and Hui Song. Towards a model-based serverless platform for the cloud-edge-iot continuum. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 851–858, 2022. (Cited at page 42)

[70] Fabiana Rossi, Simone Falvo, and Valeria Cardellini. Gofs: Geo-distributed scheduling in openfaas. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2021. (Cited at page 42)

[71] Yang Tang and Junfeng Yang. Lambdata: Optimizing serverless computing by making data intents explicit. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 294–303, 2020. (Cited at page 42)

[72] Achilleas Tzenetopoulos, Evangelos Apostolakis, Aphrodite Tzomaka, Christos Papakostopoulos, Konstantinos Stavrakakis, Manolis Katsaragakis, Ioannis Oroutzoglou, Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. Faas and curious: Performance implications of serverless functions on edge computing platforms. In Heike Jagode, Hartwig Anzt, Hatem Ltaief, and Piotr Luszczek, editors, *High Performance*

Computing, pages 428–438, Cham, 2021. Springer International Publishing. (Cited at page 42)

[73] B. Przybylski, P. Zuk, and K. Rzadca. Data-driven scheduling in serverless computing to reduce response time. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 206–216, Los Alamitos, CA, USA, may 2021. IEEE Computer Society. (Cited at page 42)

[74] T. Melamed. Interpretation for serverless. *OWASP Top 10*, 2017. (Cited at page 58)

[75] Nuno Mateus-Coelho and Manuela Cruz-Cunha. Serverless service architectures and security minimals. In *2022 10th International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–6, 2022. (Cited at page 58)

[76] Mingyu Wu, Zeyu Mi, and Yubin Xia. A survey on serverless computing and its implications for jointcloud computing. In *2020 IEEE International Conference on Joint Cloud Computing*, volume 18, 2020. (Cited at pages 58 e 59)

[77] Jannath Nisha O.S. and S. Mary Saira Bhanu. A survey on code injection attacks in mobile cloud computing environment. In *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 1–6, 2018. (Cited at page 58)

[78] Mohammed A. Aleisa, Abdullah Abuhussein, and Frederick T. Sheldon. Access control in fog computing: Challenges and research agenda. *IEEE Access*, 8:83986–83999, 2020. (Cited at page 58)

[79] Vasudev Dehalwar, Akhtar Kalam, Mohan Lal Kolhe, and Aladin Zayegh. Review of web-based information security threats in smart grid. In *2017 7th International Conference on Power Systems (ICPS)*, pages 849–853, 2017. (Cited at page 58)

[80] G. Mani, B. Srinivasa Rao, D. J. Santosh Kumar, and Chitturi Prasad. Distributed information flow control in serverless computing. In *2022 4th International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2022. (Cited at page 58)

[81] Valentin Vallois, Fouad Guenane, and Ahmed Mehaoua. Reference architectures for security-by-design iot: Comparative study. In *2019 Fifth Conference on Mobile and Secure Services (MobiSecServ)*, pages 1–6, 2019. (Cited at page 59)

[82] Feras M. Awaysheh, Mohammad N. Aladwan, Mamoun Alazab, Sadi Alawadi, José C. Cabaleiro, and Tomás F. Pena. Security by design for big data frameworks over cloud

computing. *IEEE Transactions on Engineering Management*, 69(6):3676–3693, 2022. (Cited at page 59)

[83] Tiago Espinha Gasiba and Ulrike Lechner. Raising secure coding awareness for software developers in the industry. In *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, pages 141–143, 2019. (Cited at page 59)

[84] W. O'Meara and R. G. Lennon. Serverless computing security: Protecting application logic. *31st Irish Signals and Systems Conference (ISSC)*, 2020. (Cited at page 59)

[85] Xing Li, Xue Leng, and Yan Chen. Securing serverless computing: Challenges, solutions, and opportunities. In *IEEE Network*, 2022. (Cited at page 59)

[86] Andrea Sabbioni, Carlo Mazzocca, Michele Colajanni, Rebecca Montanari, and Antonio Corradi. A fully decentralized architecture for access control verification in serverless environments. In *2022 IEEE Symposium on Computers and Communications (ISCC)*, 2022. (Cited at page 59)

[87] Muhammed Golec, Ridvan Ozturac, Zahra Pooranian, Sukhpal Singh Gill, and Rajkumar Buyya. ifaasbus: A security- and privacy-based lightweight framework for serverless computing using iot and machine learning. In *IEEE Transactions on Industrial Informatics*, volume 18, pages 3522–3529, 2021. (Cited at page 59)

[88] Dhouha Ayed, Paul-Andrei Dragan, Edith Félix, Zoltán Adám Mann, Eliot Salant, Robert Seidl, Anestis Sidiropoulos, Steve Taylor, and Ricardo Vitorino. Protecting sensitive data in the cloud-to-edge continuum: The fogprotect approach. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022. (Cited at page 59)

[89] J. Barr. Firecracker lightweight virtualization for serverless computing. 2018. (Cited at page 59)

[90] Google. gvisor: A container sandbox runtime focused on security efficiency and ease of use. 2019. (Cited at page 59)

[91] Hamza Javed, Adel N. Toosi, and Mohammad S. Aslanpour. *Serverless Platforms on the Edge: A Performance Analysis*, pages 165–184. Springer International Publishing, Cham, 2022. (Cited at page 60)

[92] Mohammad Sadegh Aslanpour, Adel N. Toosi, Muhammad Aamir Cheema, and Raj Gaire. Energy-aware resource scheduling for serverless edge computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 190–199, 2022. (Cited at page 60)

[93] Minh Nguyen, Saptarshi Debroy, Prasad Calyam, Zhen Lyu, and Trupti Joshi. Multi-cloud performance and security-driven brokering for bioinformatics workflows. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, 2019. (Cited at page 60)

[94] Georgios L. Stavrinides and Helen D. Karatza. Security and cost aware scheduling of real-time iot workflows in a mist computing environment. In *2021 8th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2021. (Cited at page 60)

[95] Yu Liu, Dapeng Lan, Zhibo Pang, Magnus Karlsson, and Shaofang Gong. Performance evaluation of containerization in edge-cloud computing stacks for industrial applications: A client perspective. *IEEE Open Journal of the Industrial Electronics Society*, 2:153–168, 2021. (Cited at page 72)

[96] Thomas Rausch, Schahram Dustdar, and Rajiv Ranjan. Osmotic message-oriented middleware for the internet of things. *IEEE Cloud Computing*, 5(2):17–25, 2018. (Cited at page 75)

[97] Anelis Pereira-Vale, Gastón Márquez, Hernán Astudillo, and Eduardo B. Fernandez. Security mechanisms used in microservices-based systems: A systematic mapping. *Proceedings - 2019 45th Latin American Computing Conference, CLEI 2019*, sep 2019. (Cited at page 76)

[98] Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20, 2018. (Cited at page 76)

[99] A. Banati, E. Kail, K. Karoczkai, and M. Kozlovszky. Authentication and authorization orchestrator for microservice-based software architectures. *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018 - Proceedings*, pages 1180–1184, 6 2018. (Cited at page 76)

[100] Dimitrios Kallergis, Zacharenia Garofalaki, Georgios Katsikogiannis, and Christos Douligeris. Capodaz: A containerised authorisation and policy-driven architecture using microservices. *Ad Hoc Networks*, 104, 7 2020. (Cited at page 77)

[101] Daniel Richter, Tim Neumann, and Andreas Polze. Security considerations for microservice architectures. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018-January:608–615, 2018. (Cited at page 77)

[102] Davy Preuveneers and Wouter Joosen. Access control with delegated authorization policy evaluation for data-driven microserviceworkflows. *Future Internet*, 9, 9 2017. (Cited at page 77)

[103] Anelis Pereira-Vale, Eduardo B. Fernandez, Raúl Monge, Hernán Astudillo, and Gastón Márquez. Security in microservice-based systems: A multivocal literature review. *Computers and Security*, 103, 4 2021. (Cited at page 77)

[104] Alina Buzachis and Massimo Villari. Basic principles of osmotic computing: Secure and dependable microelements (mels) orchestration leveraging blockchain facilities. *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 2019-June:47–52, 1 2019. (Cited at page 78)

[105] Lorenzo Carnevale, Armando Ruggeri, Francesco Martella, Antonio Celesti, Maria Fazio, and Massimo Villari. Multi hop reconfiguration of end-devices in heterogeneous edge-iot mesh networks. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2021. (Cited at page 78)

[106] Pawel Szalachowski. Password-authenticated decentralized identities. *IEEE Transactions on Information Forensics and Security*, 16:4801–4810, 2021. (Cited at page 78)

[107] Meng Kang and Victoria Lemieux. A decentralized identity-based blockchain solution for privacy-preserving licensing of individual-controlled data to prevent unauthorized secondary data usage. *Ledger*, 6, 11 2021. (Cited at page 78)

[108] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. A survey on essential components of a self-sovereign identity. *Computer Science Review*, 30:80–86, 11 2018. (Cited at page 78)

[109] Komal Gilani, Emmanuel Bertin, Julien Hatin, and Noel Crespi. A survey on blockchain-based identity management and decentralized privacy for personal data. *2020 2nd*

Conference on Blockchain Research and Applications for Innovative Networks and Services, BRAINS 2020, pages 97–101, 9 2020. (Cited at page 78)

[110] Kubernetes DaemonSet, howpublished = `https://cloud.google.com/kubernetes-engine/docs/concepts/daemonset`, note = Last access May 2021. (Cited at page 84)

[111] Antonino Galletta, Armando Ruggeri, Maria Fazio, Gianluca Dini, and Massimo Villari. Mesmart-pro: Advanced processing at the edge for smart urban monitoring and reconfigurable services. *Journal of Sensor and Actuator Networks*, 9(4), 2020. (Cited at page 86)

[112] J. Kalajdjieski, B. R. Stojkoska, and K. Trivodaliev. Iot based framework for air pollution monitoring in smart cities. In *2020 28th Telecommunications Forum (TELFOR)*, pages 1–4, 2020. (Cited at page 89)

[113] Jeong-Min Seo, Haanju Yoo, Kimin Yun, Hyunil Kim, and Sang-Il Choi. Behavior recognition of a person in a daily video using joint position information. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 172–174, 2018. (Cited at page 89)

[114] Duangduen Asavasuthirakul and Hassan Karimi. Comparative evaluation and analysis of online geocoding services. *International Journal of Geographical Information Science*, 24:1081–1100, 06 2010. (Cited at page 90)

[115] Mapcode, howpublished = `https://www.mapcode.com/about`, note = Last access May 2021. (Cited at page 90)

[116] What3Word, howpublished = `https://what3words.com/about-us/`, note = Last access May 2021. (Cited at page 90)

[117] Alejandro Gómez-Cárdenas, Xavi Masip-Bruin, Eva Marín-Tordera, and Sarang Kahvazadeh. A novel and scalable naming strategy for iot scenarios. In Kohei Arai, Rahul Bhatia, and Supriya Kapoor, editors, *Proceedings of the Future Technologies Conference (FTC) 2018*, pages 122–133, Cham, 2019. Springer International Publishing. (Cited at page 91)

[118] A. Longo, A. De Matteis, and M. Zappatore. Urban pollution monitoring based on mobile crowd sensing: An osmotic computing approach. In *2018 IEEE 4th International*

Conference on Collaboration and Internet Computing (CIC), pages 380–387, 2018. (Cited at page 91)

[119] B. Filocamo, A. Galletta, M. Fazio, J. A. Ruiz, M. A. Sotelo, and M. Villari. An innovative osmotic computing framework for self adapting city traffic in autonomous vehicle environment. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 01267–01270, 2018. (Cited at page 91)

[120] A. Souza, Z. Wen, N. Cacho, A. Romanovsky, P. James, and R. Ranjan. Using osmotic services composition for dynamic load balancing of smart city applications. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 145–152, 2018. (Cited at page 91)

[121] S. Hati and D. De. Obsc:osmotic blockchain based framework for smart city environment. In *2020 Fifth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, pages 143–148, 2020. (Cited at page 92)

[122] A. Galletta, M. Fazio, A. Celesti, and M. Villari. On the applicability of secret share algorithms for osmotic computing. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2020. (Cited at page 92)

[123] G. Lencse. Benchmarking authoritative dns servers. *IEEE Access*, 8:130224–130238, 2020. (Cited at page 92)

[124] Z. Gao and A. Venkataramani. Measuring update performance and consistency anomalies in managed dns services. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 2206–2214, 2019. (Cited at page 92)

[125] C. Hesselman, M. Kaeo, L. Chapin, K. Claffy, M. Seiden, D. McPherson, D. Piscitello, A. McConachie, T. April, J. Latour, and R. Rasmussen. The dns in iot: Opportunities, risks, and challenges. *IEEE Internet Computing*, 24(4):23–32, July 2020. (Cited at page 92)

[126] Biao Zhou, A. Tiwari, Konglin Zhu, You Lu, M. Gerla, A. Ganguli, B. Shen, and D. Krzysiak. Geo-based inter-domain routing (gidr) protocol for manets. In *MILCOM 2009 - 2009 IEEE Military Communications Conference*, pages 1–7, 2009. (Cited at page 92)

[127] W. Thongthavorn and P. Rattanatamrong. Multi-container application migration with load balanced and adaptive parallel tcp. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 55–62, 2019. (Cited at page 93)

[128] D. Fernando, P. Yang, and H. Lu. Sdn-based order-aware live migration of virtual machines. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 1818–1827, 2020. (Cited at page 93)

[129] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 178–185, 2018. (Cited at page 111)

[130] Lorenzo Civolani, Guillaume Pierre, and Paolo Bellavista. Fogdocker: Start container now, fetch image later. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC'19, page 51–59, New York, NY, USA, 2019. Association for Computing Machinery. (Cited at page 111)

[131] Auday Al-Dulaimy, Wassim Itani, Javid Taheri, and Maha Shamseddine. bwslicer: A bandwidth slicing framework for cloud data centers. *Future Generation Computer Systems*, 112:767–784, 2020. (Cited at page 114)

[132] Wei Du, Xiran Zhang, Qiang He, Wei Liu, Guangming Cui, Feifei Chen, Yuan Ji, Chenran Cai, and Yanchao Yang. Fault-tolerating edge computing with server redundancy based on a variant of group degree centrality. In *International Conference on Service-Oriented Computing*, pages 198–214. Springer, 2020. (Cited at page 114)

[133] Fei Tao, Lin Zhang, Yongkui Liu, Ying Cheng, Lihui Wang, and Xun Xu. Manufacturing service management in cloud manufacturing: overview and future research directions. *Journal of Manufacturing Science and Engineering*, 137(4), 2015. (Cited at page 114)

[134] Benay Ray, Avirup Saha, Sunirmal Khatua, and Sarbani Roy. Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment. *IEEE Transactions on Cloud Computing*, 2020. (Cited at page 115)

[135] Bashir Mohammed, Mariam Kiran, Kabiru M Maiyama, Mumtaz M Kamala, and Irfan-Ullah Awan. Failover strategy for fault tolerance in cloud computing environment. *Software: Practice and Experience*, 47(9):1243–1274, 2017. (Cited at page 115)

[136] Jialei Liu, Shangguang Wang, Ao Zhou, Sathish AP Kumar, Fangchun Yang, and Rajkumar Buyya. Using proactive fault-tolerance approach to enhance cloud service reliability. *IEEE Transactions on Cloud Computing*, 6(4):1191–1202, 2016. (Cited at page 115)

[137] Ahmad Sharif, Mohsen Nickray, and Ali Shahidinejad. Fault-tolerant with load balancing scheduling in a fog-based iot application. *IET Communications*, 14(16):2646–2657, 2020. (Cited at page 115)

[138] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. Pregan: Preemptive migration prediction network for proactive fault-tolerant edge computing. *arXiv preprint arXiv:2112.02292*, 2021. (Cited at page 115)

[139] Yinan Wu, Gongzhuang Peng, Hongwei Wang, and Heming Zhang. A two-stage fault tolerance method for large-scale manufacturing network. *IEEE Access*, 7:81574–81592, 2019. (Cited at page 115)

[140] Asad Javed, Keijo Heljanko, Andrea Buda, and Kary Främling. Cefiot: A fault-tolerant iot architecture for edge and cloud. In *2018 IEEE 4th world forum on internet of things (WF-IoT)*, pages 813–818. IEEE, 2018. (Cited at page 115)

[141] Kolade Olorunnife, Kevin Lee, and Jonathan Kua. Automatic failure recovery for container-based iot edge applications. *Electronics*, 10(23):3047, 2021. (Cited at page 115)

[142] Tingyan Long, Peng Chen, Yunni Xia, Ning Jiang, Xu Wang, and Mei Long. A novel fault-tolerant approach to web service composition upon the edge computing environment. In *International Conference on Web Services*, pages 15–31. Springer, 2021. (Cited at page 115)

[143] Mohammed Amoon. A framework for providing a hybrid fault tolerance in cloud computing. In *2015 Science and Information Conference (SAI)*, pages 844–849. IEEE, 2015. (Cited at page 116)

[144] Yogesh Sharma, Weisheng Si, Daniel Sun, and Bahman Javadi. Failure-aware energy-efficient vm consolidation in cloud computing systems. *Future Generation Computer Systems*, 94:620–633, 2019. (Cited at page 116)

[145] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135. IEEE, 2015. (Cited at page 116)

[146] Antonio Celesti, Davide Mulfari, Antonino Galletta, Maria Fazio, Lorenzo Carnevale, and Massimo Villari. A study on container virtualization for guarantee quality of

service in cloud-of-things. *Future Generation Computer Systems*, 99:356 – 364, 2019. (Cited at page 116)

[147] Yogesh Awasthi and Amin Salih Mohammed. Iot- a technological boon in natural disaster prediction. In *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 318–322, 2019. (Cited at page 136)

[148] Razin Farhan Hussain, Mohsen Amini Salehi, Anna Kovalenko, Yin Feng, and Omid Semiari. Federated edge computing for disaster management in remote smart oil fields. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 929–936, 2019. (Cited at page 136)

[149] Venita Babu and Vishnu Rajan. Flood and earthquake detection and rescue using iot technology. In *2019 International Conference on Communication and Electronics Systems (ICCES)*, pages 1256–1260, 2019. (Cited at page 136)

[150] Daniel Balouek-Thomert, Pedro Silva, Kevin Fauvel, Alexandru Costan, Gabriel Antoniu, and Manish Parashar. Mdsc: Modelling distributed stream processing across the edge-to-cloud continuum. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '21, New York, NY, USA, 2022. Association for Computing Machinery. (Cited at page 136)

[151] Finn Løvholt, Stefano Lorito, Jorge Macias, Manula Volpe, Jacopo Selva, and Steven Gibbons. Urgent tsunami computing. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, pages 45–50, 2019. (Cited at page 137)

[152] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018. (Cited at page 137)

[153] Yogesh S. Lonkar, Abhinav Sudhakar Bhagat, and Sd Aasif Sd Manjur. Smart disaster management and prevention using reinforcement learning in iot environment. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 35–38, 2019. (Cited at page 137)

[154] Tausifa Jan Saleem and Mohammad Ahsan Chishti. Deep learning for the internet of things: Potential benefits and use-cases. *Digital Communications and Networks*, 7(4):526–542, 2021. (Cited at page 137)

[155] Megha Vamsi Kiran Choda, Sri Vardhan Perla, Brahmender Shaik, Yuva Teja Anirudh Yelchuru, and Prasanth Yalla. A critical survey on real-time traffic sign recognition by using cnn machine learning algorithm. In *2023 International Conference on Intelligent Data Communication Technologies and Internet of Things (IDCIoT)*, pages 445–450, 2023. (Cited at page 137)

[156] Christoph Augenstein, Norman Spangenberg, and Bogdan Franczyk. Applying machine learning to big data streams : An overview of challenges. In *2017 IEEE 4th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, pages 25–29, 2017. (Cited at page 137)

[157] Siew Hoon Leong and Dieter Kranzlmüller. Towards a general definition of urgent computing. *Procedia Computer Science*, 51:2337–2346, 2015. International Conference On Computational Science, ICCS 2015. (Cited at page 137)

[158] Brandon Posey, Adam Deer, Wyatt Gorman, Vanessa July, Neeraj Kanhere, Dan Speck, Boyd Wilson, and Amy Apon. On-demand urgent high performance computing utilizing the google cloud platform. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, pages 13–23, 2019. (Cited at page 137)

[159] Zhiming Zhao, Paul Martin, Junchao Wang, Ari Taal, Andrew Jones, Ian Taylor, Vlado Stankovski, Ignacio Garcia Vega, George Suciu, Alexandre Ulisses, and Cees de Laat. Developing and operating time critical applications in clouds: The state of the art and the switch approach. *Procedia Computer Science*, 68:17–28, 2015. 1st International Conference on Cloud Forward: From Distributed to Complete Computing. (Cited at page 137)

[160] Spiros Koulouzis, Paul Martin, Huan Zhou, Yang Hu, Junchao Wang, Thierry Carval, Baptiste Grenier, Jani Heikkinen, Cees de Laat, and Zhiming Zhao. Time-critical data management in clouds: Challenges and a dynamic real-time infrastructure planner (drip) solution. *Concurrency and Computation: Practice and Experience*, 32(16):e5269. e5269 cpe.5269. (Cited at page 137)

[161] Alessio Catalfamo, Antonio Celesti, Maria Fazio, Giovanni Randazzo, and Massimo Villari. A platform for federated learning on the edge: a video analysis use case. In *2022 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2022. (Cited at page 137)

[162] Arpit Jain and Dharm Singh Jat. An edge computing paradigm for time-sensitive applications. In *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, pages 798–803, 2020. (Cited at page 137)

[163] Pretom Roy Ovi, Emon Dey, Nirmalya Roy, and Aryya Gangopadhyay. Aris: A real time edge computed accident risk inference system. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 47–54, 2021. (Cited at page 137)

[164] Daniel Hass and Josef Spillner. Workload deployment and configuration reconciliation at scale in kubernetes-based edge-cloud continuums. In *2022 21st International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 121–128, 2022. (Cited at page 137)

[165] Moulay A. Akhloufi, Andy Couturier, and Nicolás A. Castro. Unmanned aerial vehicles for wildland fires: Sensing, perception, cooperation and assistance. *Drones*, 5(1), 2021. (Cited at page 138)

[166] Katie E Doull, Carl Chalmers, Paul Fergus, Steve Longmore, Alex K Piel, and Serge A Wich. An evaluation of the factors affecting 'poacher' detection with drones and the efficacy of Machine-Learning for detection. *Sensors (Basel)*, 21(12), June 2021. (Cited at page 138)

[167] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016. (Cited at page 141)

[168] Christian Sicari, Valeria Lukaj, Antonio Celesti, Maria Fazio, and Massimo Villari. Gavin: A new platform for enriching 3d virtual indoor navigation with social-based geotags. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2021. (Cited at page 146)

[169] Mung Chiang and Tao Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[170] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020.

[171] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[172] Schahram Dustdar. Distributed computing continuum systems. In *2022 IEEE International Conference on Services Computing (SCC)*, pages 356–356, 2022.

[173] Google. Google cloud functions, 1999.

[174] IBM. Ibm serverless functions, 2023.

[175] knative. Knative, 2023.

[176] OpenFaas. Openfaas, 2023.

[177] Kaustubh Rajendra Rajput, Chinmay Dilip Kulkarni, Byungjin Cho, Wei Wang, and In Kee Kim. Edgefaasbench: Benchmarking edge devices using serverless computing. In *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*, pages 93–103, 2022.

[178] K R Sheshadri and J Lakshmi. Qos aware faas for heterogeneous edge-cloud continuum. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 70–80, 2022.

[179] Qi Xia, Winson Ye, Zeyi Tao, Jindi Wu, and Qun Li. A survey of federated learning for edge computing: Research problems and solutions. *High-Confidence Computing*, 1(1):100008, 2021.

[180] Runyu Jin and Qirui Yang. Edgefaas: A function-based framework for edge computing, 2022.

[181] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169, 2017.

[182] Pedro Garcia Lopez, Marc Sanchez-Artigas, Gerard Paris, Daniel Barcelona Pons, Alvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of FaaS orchestration

systems. *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, pages 109–114, 2019.

[183] Urmil Bharti, Deepali Bajaj, Anita Goel, and S. C. Gupta. Sequential workflow in production serverless faas orchestration platform. pages 681–693, 2021.

[184] Matteo Nardelli, Stefan Nastic, Schahram Dustdar, Massimo Villari, and Rajiv Ranjan. Osmotic Flow: Osmotic Computing + IoT Workflow. *IEEE Cloud Computing*, 4(2):68–75, 2017.

[185] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 25–30, New York, NY, USA, 2019. Association for Computing Machinery.

[186] Tyler J. Skluzacek, Ryan Chard, Ryan Wong, Zhuozhao Li, Yadu Babuji, Logan Ward, Ben Blaiszik, Kyle Chard, and Ian Foster. Serverless workflows for indexing large scientific data. pages 43–48, 12 2019.

[187] Wei Huang, Yizheng Zhou, and Bin Yu. Query pipeline, 04 2015.

[188] Philipp Ross and Andre Luckow. Edgeinsight: Characterizing and modeling the performance of machine learning inference on the edge and cloud. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1897–1906, 2019.

[189] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. Towards serverless as commodity: A case of knative. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 13–18, New York, NY, USA, 2019. Association for Computing Machinery.

[190] Andrea Morichetta, Victor Casamayor Pujol, and Schahram Dustdar. A roadmap on learning and reasoning for distributed computing continuum ecosystems. In *2021 IEEE International Conference on Edge Computing (EDGE)*, pages 25–31, 2021.

[191] Daniel Rosendo, Alexandru Costan, Gabriel Antoniu, Matthieu Simonin, Jean-Christophe Lombardo, Alexis Joly, and Patrick Valduriez. Reproducible performance optimization of complex applications on the edge-to-cloud continuum. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 23–34, 2021.

[192] Chung-Hyun Ahn, Tae-Hyun Hwang, and Kyoung-Ho Choi. A framework of augmented reality for geotagged videos. In *2015 21st Korea-Japan Joint Workshop on Frontiers of Computer Vision (FCV)*, pages 1–4, 2015.

[193] Li Shengyi and Wang Jia. Research on integrated application of virtual reality technology based on bim. In *2016 Chinese Control and Decision Conference (CCDC)*, pages 2865–2868, 2016.

[194] J. Liu, J. Wan, B. Zeng, Q. Wang, H. Song, and M. Qiu. A scalable and quick-response software defined vehicular network assisted by mobile edge computing. *IEEE Communications Magazine*, 55(7):94–100, 2017.

[195] Lorenzo Carnevale, Antonio Celesti, Maria Di Pietro, and Antonino Galletta. How to conceive future mobility services in smart cities according to the fiware frontiercities experience. *IEEE Cloud Computing*, 5(5):25–36, 2018.

[196] M. Gamal, R. Rizk, H. Mahdi, and B. E. Elnaghi. Osmotic bio-inspired load balancing algorithm in cloud computing. *IEEE Access*, 7:42735–42744, 2019.

[197] M. Villari, M. Fazio, S. Dustdar, O. Rana, D. N. Jha, and R. Ranjan. Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things. *Computer*, 52(8):14–26, 2019.

[198] S. Rasool, A. Saleem, M. Iqbal, T. Dagiuklas, A. K. Bashir, S. Mumtaz, and S. A. Otaibi. Blockchain-enabled reliable osmotic computing for cloud of things: Applications and challenges. *IEEE Internet of Things Magazine*, 3(2):63–67, 2020.

[199] David Balla, Markosz Maliosz, and Csaba Simon. Open source faas performance aspects. In *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, pages 358–364, 2020.

[200] R. Gowri Prakash, R. Shankar, and S. Duraisamy. Fupa: Future utilization prediction algorithm based load balancing scheme for optimal vm migration in cloud computing. In *2020 Fourth International Conference on Inventive Systems and Control (ICISC)*, pages 638–644, 2020.

[201] Karim Djemame, Matthew Parker, and Daniel Datsev. Open-source serverless architectures: an evaluation of apache openwhisk. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 329–335, 2020.

[202] C. Jiang and J. Wan. A thing-edge-cloud collaborative computing decision-making method for personalized customization production. *IEEE Access*, 9:10962–10973, 2021.

[203] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Exploiting geotagged resources for spatial clustering on social network services. *Concurrency Computation Practice and Experience*, 22(6):685–701, 2010.

[204] Cristiano Aguzzi, Lorenzo Gigli, Luca Sciullo, Angelo Trotta, and Marco Di Felice. From Cloud to Edge: Seamless Software Migration at the Era of the Web of Things. *IEEE Access*, 8, 2020.

[205] S. Alberternst, A. Anisimov, A. Antakli, B. Duppe, H. Hoffmann, M. Meiser, M. Muaz, D. Spieldenner, and I. Zinnikus. From things into clouds – and back. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 668–675, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[206] Auday Al-Dulaimy, Wassim Itani, Javid Taheri, and Maha Shamseddine. bwslicer: A bandwidth slicing framework for cloud data centers. *Future Generation Computer Systems*, 112:767–784, 2020.

[207] M.S. Altaf, Z. Hashisho, and M. Al-Hussein. A method for integrating occupational indoor air quality with building information modeling for scheduling construction activities. *Canadian Journal of Civil Engineering*, 41(3):245–251, 03 2014. cited By 7.

[208] Mohammad Sadegh Aslanpour, Adel Toosi, Muhammad Cheema, and Raj Gaire. Energy-aware resource scheduling for serverless edge computing. 05 2022.

[209] Aris Anagnostopoulos, Fabio Petroni, and Mara Sorella. Targeted interest-driven advertising in cities using Twitter. *Data Mining and Knowledge Discovery*, 32(3):737–763, 2018.

[210] Dimitris Apostolou, Yiannis Verginadis, and Gregoris Mentzas. In the Fog: Application Deployment for the Cloud Continuum. *IISA 2021 - 12th International Conference on Information, Intelligence, Systems and Applications*, 2021.

[211] Sepehr Alizadehsalehi, Ahmad Hadavi, and Joseph Chuenhuei Huang. Assessment of aec students' performance using bim-into-vr. *Applied Sciences*, 11(7), 2021.

[212] K. jr and B. Michalík. Laser scanning for bim and results visualization using vr. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-5/W2:49–52, 09 2019.

[213] Berta Carrión-Ruiz, Silvia Blanco-Pons, M. Duong, J. Chartrand, M. Li, Kristine Prochnau, Stephen Fai, and José Lerma. Augmented experience to disseminate cultural heritage: House of commons windows, parliament hill national historic site (canada). *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W9:243–247, 01 2019.

[214] Chongzheng Zhao and Lin Lv. Research on the feasibility of solving problems in construction of sponge city based on gis, vr and bim combined technology. *IOP Conference Series: Earth and Environmental Science*, 170:022068, 07 2018.

[215] Joy Arulraj, Abhijit Chatterjee, Alexandros Daglis, Ashutosh Dhekne, and Umakishore Ramachandran. ecloud: A vision for the evolution of the edge-cloud continuum. *Computer*, 54(5):24–33, 2021.

[216] Xuejun Ding, Yong Tian, and Yan Yu. A real-time big data gathering algorithm based on indoor wireless sensor networks for risk analysis of industrial operations. *IEEE Transactions on Industrial Informatics*, 12(3):1232–1242, 2016.

[217] Marco Viceconti, Peter Hunter, and Rod Hose. Big data, big knowledge: Big data for personalized healthcare. *IEEE Journal of Biomedical and Health Informatics*, 19(4):1209–1215, 2015.

[218] Alvaro A. Cárdenas, Pratyusa K. Manadhata, and Sreeranga P. Rajan. Big data analytics for security. *IEEE Security Privacy*, 11(6):74–76, 2013.

[219] Alessio Catalfamo, Antonio Celesti, Maria Fazio, Giovanni Randazzo, and Massimo Villari. A platform for federated learning on the edge: A video analysis use case. In *2022 IEEE Symposium on Computers and Communications (ISCC): 27th IEEE Symposium on Computers and Communications - 12th Workshop on Management of Cloud and Smart City Systems (MoCS 2022)*, 2022.

[220] C.-H. Chang, C.-Y. Lin, R.-G. Wang, and C.-C. Chou. Applying deep learning and building information modeling to indoor positioning based on sound. pages 193–199, 2019. cited By 1.

[221] Maarten Clements, Pavel Serdyukov, Arjen P. De Vries, and Marcel J.T. Reinders. Using flickr geotags to predict user travel behaviour. *SIGIR 2010 Proceedings - 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, (May 2014):851–852, 2010.

[222] Jeremy W. Crampton, Mark Graham, Ate Poorthuis, Taylor Shelton, Monica Stephens, Matthew W. Wilson, and Matthew Zook. Beyond the geotag: Situating 'big data' and leveraging the potential of the geoweb. *Cartography and Geographic Information Science*, 40(2):130–139, 2013.

[223] Feng Chen, Pan Deng, Jiafu Wan, Daqiang Zhang, Athanasios V. Vasilakos, and Xiaohui Rong. Data mining for the internet of things: Literature review and challenges. *International Journal of Distributed Sensor Networks*, 11(8):431047, 2015.

[224] C. Savaglio, P. Gerace, G. Di Fatta, and G. Fortino. Data mining at the iot edge. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2019.

[225] Claudio Savaglio and Giancarlo Fortino. A simulation-driven methodology for iot data mining based on edge computing. *ACM Trans. Internet Technol.*, 21(2), March 2021.

[226] Ruofei Du and David Li. Geollery : A Mixed Reality Social Media Platform Geollery : A Mixed Reality Social Media Platform. (April), 2019.

[227] Open Manufacturing Platform (OMP). Edge computing in the context of open manufacturing. `https://open-manufacturing.org/wp-content/uploads/sites/101/2021/07/OMP-IIoT-Connectivity-Edge-Computing-20210701.pdf`, 2021. Accessed: 2021-10-20.

[228] etcd, howpublished = `https://etcd.io/`, note = Last access May 2021.

[229] M. Fu and R. Liu. Automatic generation of path networks for evacuation using building information modeling. pages 320–327, 2019. cited By 4.

[230] S. Hong, J. Jung, S. Kim, H. Cho, J. Lee, and J. Heo. Semi-automated approach to indoor mapping for 3d as-built building information modeling. *Computers, Environment and Urban Systems*, 51:34–46, 2015. cited By 63.

[231] Wolfgang Hürst, Kevin Ouwehand, Marijn Mengerink, Aaron Duane, and Cathal Gurrin. Geospatial access to lifelogging photos in virtual reality. *LSC 2018 - Proceedings*

of the 2018 ACM Workshop on the Lifelog Search Challenge, co-located with ICMR 2018, (June):33–37, 2018.

[232] Paulo Flores. *Global and Local Coordinates*, volume 168. 03 2015.

[233] R. Ivanov. An approach for developing indoor navigation systems for visually impaired people using building information modeling. *Journal of Ambient Intelligence and Smart Environments*, 9(4):449–467, 2017. cited By 12.

[234] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 423–424, 2004.

[235] Slava Kisilevich, Milos Krstajic, Daniel Keim, Natalia Andrienko, and Gennady Andrienko. Event-based analysis of people's activities and behavior using Flickr and Panoramio geotagged photo collections. *Proceedings of the International Conference on Information Visualisation*, pages 289–296, 2010.

[236] Kubernetes, howpublished = `https://kubernetes.io/`, note = Last access May 2021.

[237] J.-K. Lee, J. Shin, and Y. Lee. Circulation analysis of design alternatives for elderly housing unit allocation using building information modelling-enabled indoor walkability index. *Indoor and Built Environment*, 29(3):355–371, 2020. cited By 2.

[238] Agnieszka Leszczynski and Jeremy Crampton. Introduction: Spatial Big Data and everyday life. *Big Data and Society*, 3(2):1–6, 2016.

[239] Y.-W. Lim. Building information modeling for indoor environmental performance analysis. *American Journal of Environmental Sciences*, 11(2):55–61, 2015. cited By 11.

[240] Bertram Ludäscher, Shawn Bowers, and Timothy McPhillips. *Scientific Workflows*, pages 2507–2511. Springer US, Boston, MA, 2009.

[241] Takashi Nicholas Maeda, Mitsuo Yoshida, Fujio Toriumi, and Hirotada Ohashi. Extraction of tourist destinations and comparative analysis of preferences between foreign tourists and domestic tourists on the basis of geotagged social media data. *ISPRS International Journal of Geo-Information*, 7(3), 2018.

[242] E. Martins Taiwo, K. Bin Yahya, and Z. Haron. Utilisation of building information modelling for indoor environmental quality assessment - a review. volume 220, 2019. cited By 0.

[243] Benazir Neha, Sanjaya Kumar Panda, Pradip Kumar Sahu, Kshira Sagar Sahoo, and Amir H. Gandomi. A systematic review on osmotic computing. *ACM Trans. Internet Things*, 3(2), feb 2022.

[244] M. Villari, M. Fazio, S. Dustdar, O. Rana, D.N. Jha, and R. Ranjan. Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things. *Computer, Volume: 52*, 2019.

[245] Sunil Kumar Mohanty, Gopika Premsankar, and Mario Di Francesco. An evaluation of open source serverless computing frameworks. In *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, volume 2018-Decem, pages 115–120. IEEE Computer Society, dec 2018.

[246] Muhammad Mudassar, Yanlong Zhai, and Liao Lejian. Adaptive fault-tolerant strategy for latency aware iot application executing in edge computing environment. *IEEE Internet of Things Journal*, 2022.

[247] Yusuke Nakaji and Keiji Yanai. Visualization of real-world events with geotagged tweet photos. *Proceedings of the 2012 IEEE International Conference on Multimedia and Expo Workshops, ICMEW 2012*, pages 272–277, 2012.

[248] W. Natephra, A. Motamedi, T. Fukuda, and N. Yabuki. Integrating building information modeling and virtual reality development engines for building indoor lighting design. *Visualization in Engineering*, 5(1), 2017. cited By 33.

[249] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and Massimo Villari. Towards osmotic computing: Analyzing overlay network solutions to optimize the deployment of container-based microservices in fog, edge and iot environments. *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, 2018.

[250] Antonino Galletta, Christian Sicari, Antonio Celesti, and Massimo Villari. Oce-dns: an innovative osmotic computing enabled domain name system. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 642–648, 2021.

[251] Uwe Breitenbücher, Christian Endres, Kálmán Képes, Oliver Kopp, Frank Leymann, Sebastian Wagner, Johannes Wettinger, and Michael Zimmermann. The opentosca ecosystem – concepts & tools. *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016*, pages 112–130, 2016.

[252] Openwhisk Workflow, howpublished = `https://github.com/apache/openwhisk-composer`, note = Last access May 2022.

[253] Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302, 2007.

[254] Saed Sayad. Real time data mining. 01 2017.

[255] Serverless workflows for containerised applications in the cloud continuum. *Journal of Grid Computing*, 19, 2021.

[256] Fei Chen, Tao Xiang, Yuanyuan Yang, and Sherman S.M. Chow. Secure cloud storage meets with secure network coding. *IEEE Transactions on Computers*, 65(6):1936–1948, 2016.

[257] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 2019.

[258] L. F. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. Silva, C. Lee, and O. Rana. The internet of things, fog and cloud continuum: Integration and challenges. *Internet of Things 3-4*, 2018.

[259] Taylor Shelton. Spatialities of data: mapping social media 'beyond the geotag'. *GeoJournal*, 82(4):721–734, 2017.

[260] Luke Sloan and Jeffrey Morgan. Who tweets with their location? Understanding the relationship between demographic characteristics and the use of geoservices and geotagging on twitter. *PLoS ONE*, 10(11):1–15, 2015.

[261] Ruofei Du and Amitabh Varshney. Social Street View: Blending Immersive Street Views with Geo-Tagged Social Media. pages 77–85, 2016.

[262] Yi Cheng Song, Yong Dong Zhang, Juan Cao, Tian Xia, Wu Liu, and Jin Tao Li. Web video geolocation by geotagged social resources. *IEEE Transactions on Multimedia*, 14(2):456–470, 2012.

[263] Jong-Won Lee, Deuk-Woo Kim, Seung-Eon Lee, and Jae-Weon Jeong. Development of a building occupant survey system with 3d spatial information. *Sustainability*, 12(23), 2020.

[264] Y. Tan, Y. Fang, T. Zhou, Q. Wang, and J.C.P. Cheng. Improve indoor acoustics performance by using building information modeling. pages 959–966, 2017. cited By 5.

[265] K. Tse, A. Wong, and F. Wong. Design visualisation and documentation with building information modelling - a case study. pages 241–248, 2007. cited By 0.

[266] F. Vittori, I. Pigliautile, and A.L. Pisello. Subjective thermal response driving indoor comfort perception: A novel experimental analysis coupling building information modelling and virtual reality. *Journal of Building Engineering*, 41, 2021. cited By 0.

[267] Lina Zhong, Liyu Yang, Jia Rong, and Haoyu Kong. A Big Data Framework to Identify Tourist Interests Based on Geotagged Travel Photos. *IEEE Access*, 8:85294–85308, 2020.

[268] X. Zhou, Q. Xie, M. Guo, J. Zhao, and J. Wang. Accurate and efficient indoor pathfinding based on building information modeling data. *IEEE Transactions on Industrial Informatics*, 16(12):7459–7468, 2020. cited By 2.

[269] Niranjan Suri, Zbigniew Zielinski, Mauro Tortonesi, Christoph Fuchs, Manas Pradhan, Konrad Wrona, Janusz Furtak, Dragos Bogdan Vasilache, Michael Street, Vincenzo Pellegrini, Giacomo Benincasa, Alessandro Morelli, Cesare Stefanelli, Enrico Casini, and Michal Dyk. Exploiting smart city iot for disaster recovery operations. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 458–463, 2018.

[270] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud continuum: The definition. *IEEE Access*, 10:131876–131886, 2022.

[271] Mingliu Liu, Deshi Li, Huaqing Wu, Feng Lyu, and Xuemin Sherman Shen. Cooperative edge-cloud caching for real-time sensing big data search in vehicular networks. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, 2021.

[272] Sheshadri K R and J Lakshmi. Qos aware faas platform. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 812–819, 2021.

[273] Lorenzo Carnevale, Antonino Galletta, Maria Fazio, Antonio Celesti, and Massimo Villari. Designing a fiware cloud solution for making your travel smoother: The fliware experience. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 392–398, 2018.