ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

Ciclo 37

Settore Concorsuale: 01/B1 - INFORMATICA

Settore Scientifico Disciplinare: INF/01 - INFORMATICA

DECLARATIVE APPROACHES FOR CUSTOM CLOUD-EDGE SERVERLESS FUNCTION SCHEDULING

Presentata da: Giuseppe De Palma

Coordinatore Dottorato

Ilaria Bartolini

Supervisore

Gianluigi Zavattaro

Co-Supervisore

Saverio Giallorenzo

Esame finale anno 2025

Borsa di dottorato del Programma Operativo Nazionale Ricerca e Innovazione 2014-2020 (CCI 2014IT16M2OP005), risorse FSE REACT-EU, Azione IV.4 “Dottorati e contratti di ricerca su tematiche dell’innovazione” e Azione IV.5 “Dottorati su tematiche Green.” J35F21003070006

Abstract

Serverless computing has rapidly evolved from cloud-centric paradigms to embrace private edge and hybrid cloud-edge systems, addressing latency and resource optimization challenges. However, mainstream serverless platforms typically rely on rigid, hardcoded scheduling policies that fail to support the diverse functional, topological, and performance constraints required by modern applications. In particular cloud-edge serverless applications, or serverless deployments, spanning multiple regions introduce the need to govern the scheduling of functions to satisfy their functional constraints or avoid performance degradation to meet user-defined goals. We address the problem of *function-execution scheduling* in this thesis by first proposing a declarative language of *Allocation Priority Policies* (APP), enabling developers to specify scheduling policies tailored to their application needs, and we show an implementation of APP on top of Apache OpenWhisk, validated with a cloud-edge use case. Building on this foundation, the focus shifts to topology-aware scheduling through the development of tAPP, a language extension capable of enforcing co-existing topological constraints across hybrid deployments without requiring custom platform modifications. We prove our approach feasible by implementing a tAPP-based Apache OpenWhisk, and show that our extension naturally allows for cloud-edge deployments with topology-aware requirements which cannot be supported by standard deployments of vanilla OpenWhisk. We then focus on *affinity-aware* scenarios, i.e., where, for performance and functional requirements, the allocation of a function depends on the presence/absence of other functions on nodes. We further extend APP to aAPP, a language that allows users to capture affinity-aware scheduling policies. An aAPP-based prototype shows that affinity constraints can be expressed and enforced with negligible overhead, enabling performance improvements where affinity matters..

This family of languages unlocks the capability for customizable scheduling in FaaS, making it possible to enforce ad-hoc optimizations in serverless applications. However, defining the "right" scheduling policy is far from trivial, often requiring rounds of refinement that involve knowledge of the underlying infrastructure, guesswork, and empirical testing. We start investigating how information derived from static analysis could be incorporated into APP scheduling function policies to help users select the best-performing workers at function allocation. To this end, we develop a cost-variant APP called cAPP, which incorporates a pipeline capable of extracting cost equations

from functions' code, synthesizing cost expressions through the usage of off-the-shelf solvers, and extending APP policies to consider this information.

Finally, recognizing the resource-constrained nature of cloud-edge scenarios, we present FunLess, a FaaS platform capable of running bare-metal on edge devices without the need for container orchestration layers. FunLess leverages WebAssembly (Wasm) as its runtime, providing a lightweight execution environment with enhanced portability and a reduced memory footprint. We also perform a comparative analysis of the energy consumption of FunLess, OpenWhisk and a container-based service architecture. Moreover, FunLess features built-in support for APP, enabling the execution of customizable scheduling policies even in resource-constrained environments.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Software development and deployment have undergone multiple changes throughout the history of computing and continues to evolve even today. From the early days of mainframes, where computing was centralized on a single machine, to multi-tier architectures with multiple physical nodes connected and accessible over a network, to the modern era of cloud computing where hardware resources are virtualized and provided as a remote service. Cloud computing has fundamentally changed how software systems are designed and its own evolutionary line with several new paradigms have emerged. Starting from only providing resources such as storage and machines, to providing software systems that can compile and run applications directly, kickstarting a transition of computation from on-premises servers to a multitude of data centers distributed across the globe. Alongside these developments, Edge computing has also emerged to address the need for latency-sensitive applications and the increasing amount of data generated by Internet of Things (IoT) devices. Moving large amounts of data to the cloud for processing is not always feasible due to bandwidth or latency constraints, so as opposed to centralizing computation in remote data centers it is, instead, distributed across devices or servers located nearer to users or data-generating endpoints. Modern solutions are now evolving toward the concept of Cloud-Edge Continuum, a model that represents a spectrum of resources aiming to integrate the large-scale centralized public clouds with distributed, near-device computational resources. The key pattern of this evolution in computing has been an abstraction over the hardware and the simplification of the deployment and management of applications, reaching this newest peak with the illustrative name

of "Serverless" Computing, where, ideally, the infrastructure is completely abstracted away, hence servers (and devices) are no longer directly managed by the user. A key component of this abstraction is the Function-as-a-Service (FaaS) model, which has been gaining more and more attention both in academic research and industry. This service model allows developers to deploy programs in the form of event-driven, stateless and ephemeral functions, which can be executed over a large number of servers automatically. The simplicity in deployment and scalability attracted many researchers in exploring its usage in different scenarios, such as investigating new ways to architect distributed systems, finding new optimizations in workload scheduling and execution, and its applicability to IoT and Edge computing. Popularized through AWS Lambda as the first FaaS platform offered by a major cloud provider, many other providers have followed suit with their own proprietary or open-source platforms. Operating on the premise of automatically allocating resources for execution offers an advantage in terms of cost efficiency for customers, but the underlying scheduling and resource allocation mechanisms are generally opaque and inflexible. These rigid mechanisms can fail to adapt to heterogeneous environments that include different types of resources, equipped with varying hardware, and that could include not only cloud data centers but also edge devices. In such distributed environments, performance optimizations require an awareness of resource locality, workload variability, and dynamic system constraints. The complexity of these challenges calls for a more flexible and expressive approach to resource management in FaaS platforms. While existing solutions often rely on predefined, black-box scheduling mechanisms, a declarative language-based approach offers the ability to precisely specify scheduling policies and reason about them systematically. This enables developers to maintain fine-grained control over function placement while abstracting away the low-level details of resource allocation. The driving motivation for the works presented in this thesis lies in exploring the optimizations achievable in FaaS platforms by allowing users to maximize usage of the resources available to them. For example, latency-sensitive applications, such as those involving real-time tasks with IoT-generated data, require their execution environments to be as close as possible to the data source. Similarly, compute-intensive workloads may need to leverage powerful cloud servers, while lightweight tasks can efficiently run on resource-constrained edge devices. Current platforms fail to consider these diverse requirements and these limitations become particularly pronounced in scenarios involving hybrid deployments. To

tackle this class of problems, this thesis introduces a family of declarative languages and companion frameworks for scheduling policies, namely Allocation Priority Policies (APP) and its extensions: Topology-aware APP (tAPP) with the investigation on integrating topology-based optimizations on function scheduling, Affinity-aware APP (aAPP) by exploring the idea of (anti-)affinity constraints between functions and Cost-aware APP (cAPP) by considering the cost of function execution. These frameworks enable developers and operators to specify detailed policies for function placement, incorporating considerations such as data locality, resource availability, and their heterogeneity.

Starting with APP (in Chapter 4), we focus on the problem of optimizing function scheduling in a heterogeneous cluster of resources. In FaaS platforms the component responsible for function execution, often called worker, and the component responsible for function scheduling, i.e. choosing a worker for a function to run on, are typically distributed across one or more data centers. The open-source platforms that have become popular over the years, such as Apache OpenWhisk [8] and OpenFaaS [80], make use of simplistic scheduling approaches such as pseudo-random or round-robin selection, which do not consider the computational power of the workers or the functions they are currently executing. Not all workers are equal when allocating functions. Indeed, effects like data locality [47]—due to high latencies to access data—or session locality [47]—due to the need to authenticate and open new sessions to interact with other services—can sensibly increase the run time of functions. To tackle the challenges and opportunities for these optimizations in function scheduling we propose APP as the basis for a policy-driven, declarative scheduling language for FaaS platforms. Developers can use APP to specify a scheduling policy for their functions that the scheduler later uses to find the worker that, given the current status of the system, best fits the constraints specified by the developer of a given function. We extended the scheduler of OpenWhisk as well, to use APP-defined policies in the scheduling of serverless functions, and validated our extension with an use case combining IoT, Edge, and Cloud Computing. With the baseline ideas in place, we explored different directions in function scheduling optimizations with several APP extensions.

Following APP, we explored scenarios where FaaS platforms are deployed across multiple zones, each with its own set of workers and schedulers. As to OpenWhisk's case, platforms are deployed over a cluster of machines where, regardless of the zones and number of replicated schedulers, any worker in the cluster could be picked to run a

function. To handle topological information and cover multi-scheduler deployments, we introduce tAPP (in Chapter 5) as a first extension of APP. With tAPP we enhance the initial language with new constructs and extend OpenWhisk with new components to capture topological information at the level of workers and schedulers, and let schedulers and gateways follow tAPP policies depending on topological zones. We evaluated our tAPP-enabled OpenWhisk prototype using an Industry 4.0 case study featuring a Cloud-Edge deployment.

After tAPP, we explored the idea of affinity-aware scheduling in FaaS from observating that, at lower levels of the cloud stack, popular Infrastructure-as-a-Service (IaaS) platforms (e.g., OpenStack [82]) and Container-as-a-Service (CaaS) systems (e.g., Kubernetes [64]) allow users to express affinity and anti-affinity constraints about the allocation of VM/containers—e.g., anti-affinity constraints, to reduce overhead by shortening data paths via co-location, to increase reliability by evenly distributing VM/containers among different nodes, and for security, such as preventing the co-location of VM/containers belonging to different trust tiers. On the contrary, FaaS platforms do not natively support the possibility to express affinity-aware scheduling, where function allocation depends on the presence (affinity) or absence (anti-affinity) at scheduling time of other functions in execution on the available workers. Recognising the potential of FaaS-level affinity-aware scheduling policies, we propose aAPP (in Chapter 6) by extending APP, and we concretise our proposal by presenting a prototype implementation of an aAPP-based OpenWhisk able to enforce aAPP-defined FaaS (anti-)affinity scheduling constraints.

While studying the potential optimizations in function scheduling, we also focused on another line of research that is gaining traction in the community, the costs and sustainability of cloud computing. The advent of the cloud raised concerns about the energy consumption and environmental impact of data centers, which power cloud services, as major consumers of electricity, contributing to carbon emissions. As demand for cloud services continues to rise, optimizing resource usage and minimizing energy waste have become important challenges for both researchers and industry. By applying these concerns on our research on FaaS platforms, we propose two contributions: *i*) a novel FaaS platform, FunLess, designed to be lightweight with a focus on (mixed) edge-cloud scenarios, and *ii*) a Cost-aware extension of APP, named cAPP (in Chapter 7), with an implementation on FunLess.

With FunLess, we experimented with new ideas and more recent technologies such

as WebAssembly (Wasm) [122] to create a lightweight FaaS platform with built-in support for APP. FunLess is a new open-source platform providing decreased resource requirements, lightweight scalability, and portability via the Erlang's BEAM Virtual Machine [107] and WebAssembly for running functions. Thanks to these traits, users can run the whole FunLess distributed platform on resource-constrained edge devices without requiring a container runtime (e.g., Docker) and container-orchestration technologies (e.g., Kubernetes). We also studied the energy usage impact compared to more traditional FaaS platforms.

Regarding cAPP, we propose this new extension from the observation that if an user has knowledge about the reduced running time of a worker in performing some particular task with an external service, .e.g. in accessing a database, the user must know about the workers' topology and their latencies w.r.t. the external services used by their functions. However, users might not have such knowledge when writing their APP scripts. Moreover, the worker-service latency is a property that can dynamically change depending, e.g., on the state of the network connections, including traffic and congestion. Thus, we start investigating how information derived from static analysis could be incorporated into APP scheduling function policies to help users select the best-performing workers at function allocation. We substantiate our proposal by presenting a pipeline able to extract cost equations from functions' code, synthesising cost expressions through the usage of off-the-shelf solvers, and extending APP to consider this information. In other terms, we propose to use a combination of static analysis (applied on a function's code) and run-time monitoring (of the workers latencies in accessing the external services) to estimate a *cost* for executing a function on a worker, considering what and how it uses external services. Differently from before, the prototypical implementation of cAPP is developed on FunLess, as it is the platform we can better control which allowed us an integration of the needed components from the ground up.

Structure of the Thesis The remainder of this thesis is organized as follows. It is divided in two main parts: Background and Contributions. From the Background part, Chapter 2 provides an overview of cloud computing and its evolution over the years with a focus on the models of distribution and deployment. Chapter 3 introduces Function-as-a-Service as the core model of serverless computing and elaborates on its architectural principles, scalability features, and scheduling challenges. In the first

chapter of the Contributions part, Chapter 4, we present the design and implementation of APP, including its integration into the Apache OpenWhisk platform and the experimental evaluations. We extend the discussion to tAPP in Chapter 5, introducing its ability to support scheduling policies with topological constraints, and its integration and evaluation. In Chapter 6, we move the discussion to aAPP, presenting the new capabilities of expressing affinity and anti-affinity constraints in function scheduling, together with its implementation and evaluation. We then introduce cAPP in Chapter 7 where we go into details on the additional components and ideas built on top of APP to make scheduling aware of function running time, and we present FunLess in Chapter 8 with an overview of its architecture, design choices and a comparison on the energy usage with OpenWhisk and a classical service-oriented architecture. Finally, in Chapter 9 the conclusions are drawn and some directions for future work are discussed.

Part I

Background

Chapter 2

Cloud and Serverless Computing

2.1 The Cloud Organization

Modern-day web software is deeply integrated with services on the cloud due to the capabilities and flexibility they can offer. In this chapter, we give an introduction to cloud computing and serverless computing (the focus of this thesis) and introduce the common service models available.

Cloud computing is characterized by an easily usable and accessible pool of virtualized resources based on a pay-per-use model. Customers can acquire and release resources on demand, and when in need of scaling they can simply request more resources. Figure 2.1 shows a simplified view of the organization of cloud computing.

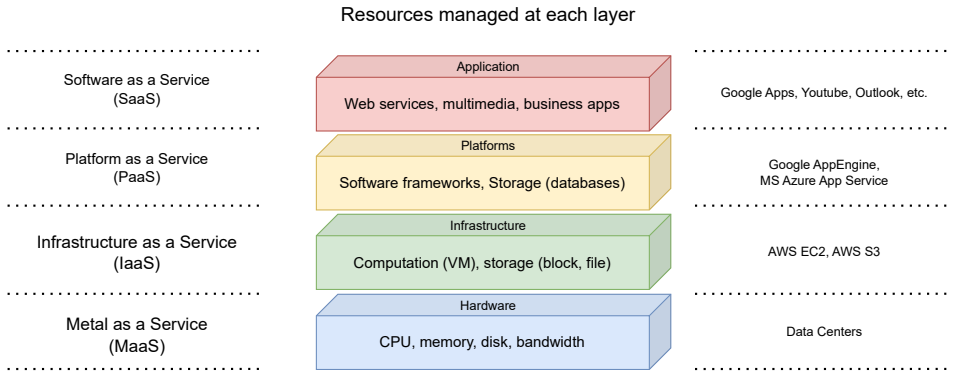


Figure 2.1: Cloud Computing Layers (adapted from Zhang et al. [130]).

We can generally divide cloud computing into four layers:

- **Hardware**: At the lowest layer, the cloud provider manages the physical hardware. Server machines, storage equipment, networking devices but also power and cooling systems. This physical layers is generally implemented in data centers, which are large buildings powering, cooling and housing all the necessary equipment to run a variety of software services.

- **Infrastructure**: On top of the hardware there are the virtualization technologies that form the backbone for most cloud computing offerings. At this level virtualization techniques are employed to provide customers an infrastructure consisting of virtual machines, virtual storage and other computing resources.

- **Platform**: Based on the virtualized resources provided by the infrastructure layer, a set of APIs and services provide the means to developers to easily develop and deploy software. In practice, an application developer is offered a vendor-specific API, which includes calls to uploading and executing a program in that vendor's cloud, giving an high-level abstraction of the underlying machines, storage and so on. For example, Amazon S3 provides an API that allows users to store locally created files in "buckets". By doing so, the file is uploaded to Amazon's cloud and can be accessed remotely. Moreover, the responsibility of conserving the file is shifted to the service provider.

- **Application**: This topmost layer is where applications are delivered to end-users over the internet. Many kinds of web applications make use of cloud computing such as web-based email, office suites (text processors, spreadsheet applications, presentation applications), collaboration tools. These applications are executed on the vendor's cloud and the vendor is responsible to make them accessible online and keep them always available without any downtime.

2.1.1 The "as-a-Service" Models Proliferation

Over the years, the cloud computing market has seen a proliferation of services that can be grouped on top of the four layers described above. They are commonly referred to as "as-a-Service" models, where the "as-a-Service" suffix means that the service is provided

over the internet based on a pay-per-use model. Starting from these four layers, cloud providers offer them to their customers as a means for outsourcing local computing infrastructures, through various interfaces (command-line tools, APIs, and web GUIs), making them their business model. This approach has led to the establishment of four main service models - MaaS, IaaS, PaaS, and SaaS - which represent increasing levels of abstraction in cloud service delivery. Fig. 2.2 shows a common representation of these service models. Numerous specialized variations have also emerged to provide specific functionalities tailored to particular needs (such as Database-as-a-Service, Storage-as-a-Service, and so on).

Metal-as-a-Service (MaaS)

The first kind of cloud service corresponds to a direct hardware (commonly called bare-metal) offering, hence the name Metal as a Service. MaaS offers to customers dedicated, bare-metal servers installed and housed by providers. Customers can choose from servers with various hardware configurations, such as different CPUs, GPUs, and memory options, to meet their specific needs. In this case, customers are responsible for configuring and maintaining the resources, while providers only handle the physical installation (and power and cooling). As an example, a company running high-performance computing tasks might want full control over the hardware and avoid user contention on shared resources.

Infrastructure-as-a-Service (IaaS)

IaaS provides virtualized computing resources over the internet, enabling customers to rent virtual machines (often referred to as "instances"), storage, and networking components on demand. This model allows customers to scale their infrastructure dynamically according to their workload requirements without the need for investing in physical hardware. Providers manage the underlying physical infrastructure and ensure its availability, while customers control and configure the operating systems and applications running on them. For example, a company developing a web application might use IaaS to quickly deploy and scale their servers based on user traffic, paying for the virtual machines only when they are needed. The pervasive use of virtualization technology revolutionized the market by providing the ability to cloud vendors to abstract from physical resources and

run workloads belonging to different customers on the same infrastructure transparently.

Platform-as-a-Service (PaaS)

This model is a step further in the abstraction of the infrastructure, further reducing the customer's effort by providing a ready-to-use execution environment on which applications can run. PaaS provides a platform (often used via APIs or web interfaces) allowing customers to develop, run, and manage applications without dealing with the underlying virtualized components (e.g., having to install all the necessary software and dependencies on virtual machines). In its purest form, PaaS is just an abstraction over the infrastructure layer with some interface to upload application code. The cloud vendor has to spin up the virtual machines, configure them, and deploy the application. This mechanics further delegates responsibility to the cloud provider, which is now responsible for the execution and scalability of the application.

Software-as-a-Service (SaaS)

At the other end of the spectrum, SaaS offers the highest level of abstraction by delivering fully functional software applications over the Internet, accessible through a web browser or mobile apps without any need for local installation or maintenance. From this perspective, SaaS customers are the end-users of the software. A wide range of business applications are offered as SaaS, such as Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), human resources management, and collaboration or productivity tools like Office 365, Google Suite or Salesforce. The main benefits of SaaS are threefold: *i)* it eliminates the need for customers to purchase, install, and maintain software, *ii)* it reduces IT management costs, *iii)* enables the customer to access the software from any device with Internet access. Usually, SaaS software is licensed on a subscription basis, with the customer paying a monthly or yearly fee to access the software.

Other "as-a-Service" Models

Based on these four main models, many other specialized services tailored to specific needs were developed. This phenomenon became known as the "Everything as a Service" (XaaS) model, where any kind of software can be offered as an online service based on a pay-per-use model. For instance, Database-as-a-Service allows customers to easily

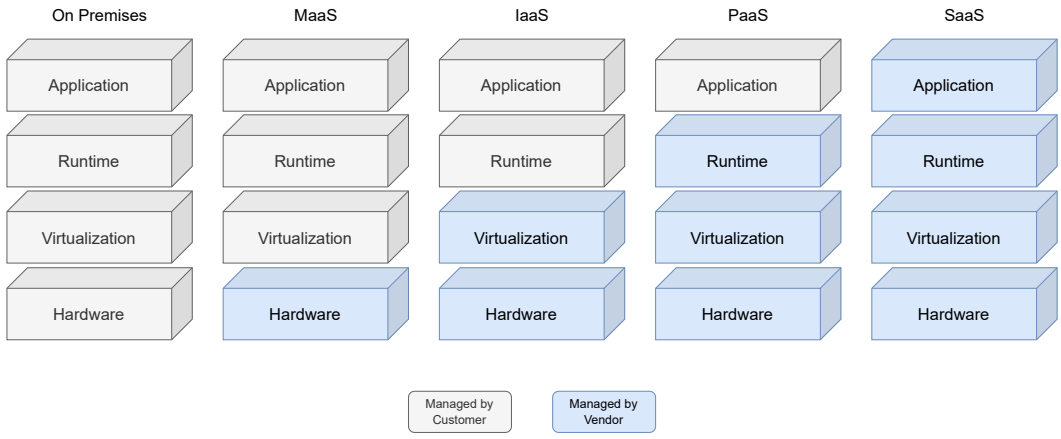


Figure 2.2: Common representation of the responsibility offloaded to the cloud vendor provided by the different Cloud Computing Service Models.

deploy, access and manage databases without the complexities of setup and maintenance. Similarly, Storage as a Service provides scalable storage solutions that can be accessed remotely, eliminating the need for on-premises hardware. This kind of services can each fall into their relative layer of the above-mentioned cloud computing organization. Of particular note, Backend-as-a-Service (BaaS) is a refinement of the PaaS model, where the cloud provider integrates in the code execution platform a set of pre-configured services, such as databases, storage, and middleware. This enhanced platform forms a complete, tighly integrated environment where the customer has access to a ready-to-use backend infrastructure. Backend-as-a-Service is one of the two components that form the basis of Serverless Computing, the other being Function-as-a-Service (FaaS) which we discuss in Chapter 3.

2.2 Types of Cloud

Cloud computing, with its several service models, has been instrumental in expanding the reach and capabilities of software applications. We can categorize the kind of cloud deployments into four main types [126, 26]: private cloud, community cloud, public cloud, and hybrid cloud. Moreover, we can consider different kinds of distribution models, such as multi-region, multi-cloud, fog computing, and edge computing.

2.2.1 Deployment Models

Private Cloud

A private cloud, also known as internal cloud, is designed for exclusive use by a single organization. It is either built and managed by the organization or outsourced to a third party. This model ensures the highest degree of control over performance, reliability, and security. However, unlike public cloud solutions, private clouds do not typically operate on a pay-as-you-go model, resembling traditional company-owned server farms in their financial structure.

Community Cloud

A community cloud is a cloud computing model where infrastructure is shared among a specific group of users, such as organizations within the same industry or with common interests. These cloud deployments are typically owned and managed by one or more organizations within the community, and they cater to the unique needs of this group. While delivering computing resources like storage, networking, servers, and applications over the Internet, community clouds are distinct from public clouds in that they are accessible only to members of the community. This model offers benefits such as cost efficiency, enhanced security, optimizations tailored to specific use cases, and compliance with regulatory requirements. By sharing resources, community clouds reduce infrastructure and management costs, providing a secure and compliant environment for the community's specialized needs.

Public Cloud

Public clouds, managed by third-party providers, offer computing resources such as storage, networking, and applications over the Internet to users worldwide. These cloud deployments operate on a pay-as-you-go model, allowing users to access and scale resources as needed without the burden of maintaining their own infrastructure. While public clouds can be cost-effective, including no initial capital investment on infrastructure, they may not provide the same level of customization as private clouds and lack fine-grained control over data, network, and security settings. Moreover, the services offered by the providers are often geared towards the so called "vendor lock-in", where customers are

tied to a specific provider due to the integrated nature of the services offered, making it difficult to migrate an application built on a specific provider's cloud to another one.

Hybrid Cloud

Hybrid clouds are a combination of the public and private cloud models that try to address the limitations of the other approaches. It allows organizations to leverage on-premises, private, and third-party public cloud services based on their specific needs. This approach enables fine-grained control over virtualized infrastructure, utilizing standardized or proprietary technology to integrate different cloud deployments. It makes it possible for organizations to use public clouds for non-sensitive tasks and private clouds for critical or sensitive workloads. However, hybrid clouds requires to carefully determine the best split of resources and workloads between the two environments.

2.2.2 Distribution Models

Cloud computing distribution models define how cloud services are deployed and accessed across different locations and platforms. A single centralized cloud approach could represent a single point of failure for the overall architecture, due to possible network connectivity problems, human errors, unpredictable failure or natural disasters. Moreover, from a business perspective, the above-mentioned phenomena of vendor lock-in can be a significant issue, making it difficult for customers to experiment with different cloud providers or migrate from one to another to cut costs or take advantage of different capabilities.

Multi-Cloud

An effective solution to cloud availability and vendor lock-in issues is adopting a Multi-Cloud strategy. Multi-Cloud involves using multiple cloud services from different providers into a single infrastructure, including public clouds like AWS, Microsoft Azure, and Google Cloud Platform, as well as private and on-premises solutions. This approach generalizes the hybrid cloud model by integrating multiple public cloud providers. Opting for Multi-Cloud helps to avoid dependency on a single provider, ensuring there are alternatives in case of outages or pricing changes. It also allows cost optimization, as different providers may offer better pricing for specific services. Implementing

Multi-Cloud requires resource management across platforms, which can be facilitated by cloud management tools and standardized technologies like OpenStack [81] and Kubernetes [63]. However, the approach may still fall short in meeting the strict locality and high data demands of numerous connected, smart services.

Multi-Region

An alternative to the Multi-Cloud strategy is the Multi-Region approach, where services are distributed across multiple geographic regions. This can also be enacted within the same cloud provider. This method enhances resilience by reducing the risk of regional failures such as natural disasters or network outages, by spreading resources across different regions. It also makes it possible to position resources in regions closer to end-users to reduce latency. Additionally, Multi-Region deployments help meeting regulatory requirements by keeping data within specific geographical boundaries. As for multi-cloud distribution, this strategy requires robust mechanisms for data synchronization, failover management, and load balancing across the regions.

Edge Computing

Going a step further in the distribution models, Edge Computing brings resources closer to the users, outside of centralized data centers typically at the edge of the network. With the ubiquitous wireless Internet access and the proliferation of IoT technology many new applications are possible in sectors such as smart cities and Industry 4.0 [12]. Traditional cloud solutions are not optimized for latency speeds, bandwidth and connectivity availability, when it comes to a decentralized network of devices connected to the cloud. As industries increasingly demand fast analysis and reaction, the delays inherent in cloud computing lead to inefficiencies and delays. Additionally, the continuos streams data generated by IoT devices require an adequate bandwidth and raise security and privacy concerns. Adding Edge Computing to the cloud continuum enables local optimizations of data processing, reducing the distance data must travel on the network and minimizing delays, by positioning storage and computing resources closer to data-producing and consuming devices. In this way, Edge Computing is perceived as a method of optimizing Cloud Computing by performing computations (such as data analytics) as close to the data sources as possible. Edge resources can be tailored to specific use

cases, providing functionalities like computing offloading, data storage, caching, and service request coordination. This proximity reduces network hops, speeding system response and interactions. New connectivity technologies like WiFi-6 and 5G further enhance network bandwidth, supporting parallel and continuous data transmission.

Fog Computing A related concept is Fog Computing, which is usually conflated with Edge Computing. Fog Computing is a layered model aiming to give ubiquitous access to a continuum of computing resources, from the cloud to the edge. The central concept is to have "fog" nodes (physical or virtual) positioned between smart end-devices and centralized cloud services [50]. These context-aware fog nodes, organized in clusters, minimize request-response times and provide local computing resources while maintaining connectivity to central services when needed. Together Edge, Fog and Cloud form the Cloud-Edge Continuum, Figure 2.3.

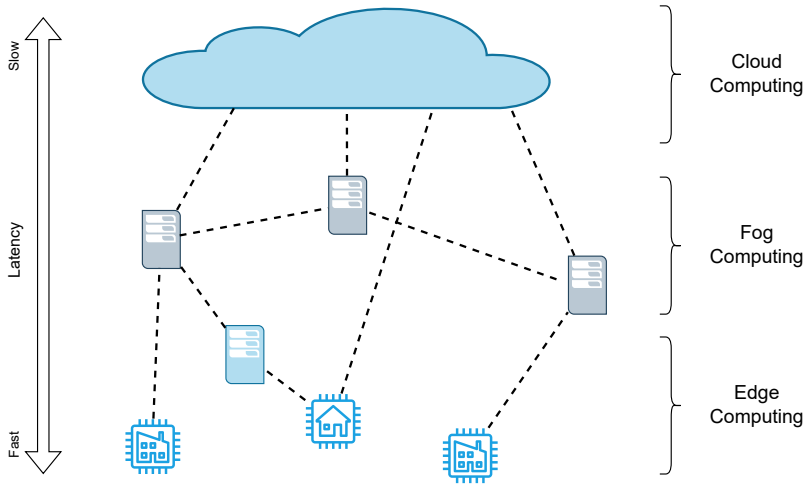


Figure 2.3: A representation of the Cloud-Edge Continuum, adapted from [50].

Cloud-Edge Continuum

Organizations can choose to adopt one or multiple cloud distribution models based on their specific needs. Multi-cloud and edge computing are emerging as promising architectural patterns that could potentially address service availability and quality demands. The development of an infrastructure beyond centralized data centers into the edge

forms the Cloud-Edge Continuum (commonly referred to with just Cloud Continuum), which combines the large-scale data processing of cloud computing with the low latency, location-aware processing of edge computing. Integrating different models within the Cloud Continuum presents challenges in orchestrating and managing these decentralized infrastructures. For an effective integration, systems are required to cooperate and coordinate across various protocols and data formats, necessitating reliable and scalable interaction mechanisms along with novel synchronization and coordination methods. This complexity can increase development costs and hinder adoption, although new service models, like serverless computing, try to abstract away the underlying resources and provide a more straightforward way to deploy applications across the continuum.

In the next chapter the focus will shift to Serverless Computing, introducing its key service model: Function-as-a-Service.

Chapter 3

Function as a Service

Function as a Service (FaaS) is another computing model that has been gaining popularity over the last few years [92, 27, 72, 44, 61, 80, 8, 59]. The main idea behind it is to have a service where a developer can register code functions together with parameters such as triggering events and data bindings. The functions are uploaded and stored in a platform that can support indipendent invocations of the functions in response to the triggering events. These FaaS platforms create an abstraction that allows users to operate as if the underlying infrastructure does not exist, therefore paired with a BaaS system to manage the FaaS platform itself, a cloud provider can offer a "Serverless" environment. With these platforms, software developers create stateless functions, that act as the basic execution unit, and develop software architectures as a composition of these functions. These compositions are often referred to as workflows or pipelines. This way of building architectures also means that users do not control where or when their code is executed. As shown in Figure 3.1, we can extend the previous service models representation from Figure 2.2 by including FaaS as an intermediate step between PaaS and SaaS. In FaaS, customers have no view of the underlying runtimes of their applications, i.e., they do not have to package their code with the necessary libraries or dependencies (or containerized) so that it can be uploaded to a PaaS system. Furthermore, they do not have to worry about scaling their applications, or introduce mechanisms to handle scaling for when the PaaS platform decides to perform replications. With FaaS, customers only have to upload code snippets (typically in the form of functions with a specific signature) and configure the events that can trigger their execution (e.g. HTTP requests). These functions can be

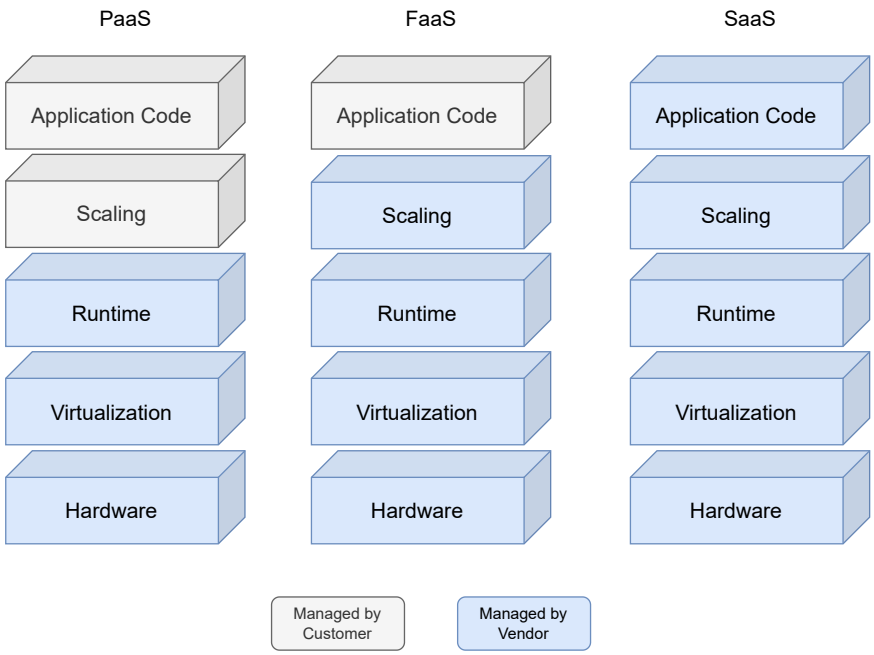


Figure 3.1: Common representation of the responsibility offloaded to the cloud vendor provided by the different Cloud Computing Service Models.

written in the platform of choice's supported programming languages and are (or should be) designed to be short-lived and stateless. Under this light, everything is managed by the cloud provider but the code itself, putting FaaS just one step below SaaS.

FaaS platforms also dynamically scale resources per request, adjusting automatically based on demand. This functionality allows for a zero-scaling feature, meaning services consume minimal resources when idle (as long as no functions are running due to some triggering, the provider just runs minimal services to listen for those triggers). There is a variety of approaches to handle functions' execution, but the most common one is to use containers in order to have an isolated environment to run the function. On a new function invocation, the platform instantiates a new container, executes the function, and then destroys the container or reuses it for future invocations. In the next section, we will discuss the various approaches to function execution. Finally, regarding this ability to spin up resources on demand to invoke functions, cloud providers have introduced a per-execution billing model, where customers pay only for function invocations or for the duration of each function execution. When using a paid FaaS offering from a cloud

vendor, it can translate into a more cost-effective solution, especially when demand of an application is low or highly variable. Instead of paying for a fixed amount (typically a monthly fee) to reserve resources, customers only pay for when their applications are in use. The per-use model can also be a double-edged sword when there are unexpected traffic spikes, or in case of a developer error that causes a function to invoke itself or other functions continuously (a common anti-pattern [95]). In both cases, the platform will silently try to meet demand by automatically scaling the functions up [111, 88].

3.1 FaaS Platforms Internals

In recent years many FaaS platforms have been developed by both industry and academia. All the major cloud providers have their own Serverless computing offerings powered by their own FaaS platforms, such as AWS Lambda, Google Cloud Functions, Azure Functions, and IBM Cloud Functions. The latter is based on Apache OpenWhisk, an open-source FaaS platform developed by IBM and later donated to the Apache Software Foundation. In the open source space there are numerous other projects, with OpenWhisk being now one of the most popular together with OpenFaaS and Knative.

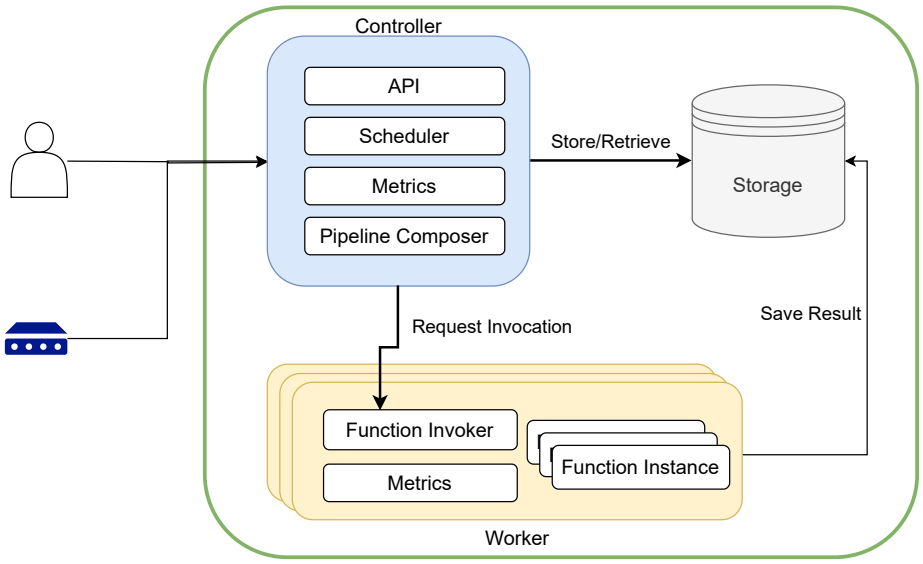


Figure 3.2: High-level FaaS architecture components.

Among all these platforms, a pattern of common architecture elements has emerged,

as shown in Figure 3.2. In particular, a Controller component responsible for managing and coordinating the functions, a Worker component (or many) that runs the functions, a storage component to store the functions, their metadata possibly their results, etc., and a triggering mechanism to map events to function invocations.

3.1.1 Core Components

Functions

The core concept for a FaaS platform is the execution and scaling of functions, which are code snippets written in some programming language supported by it and uploaded by customers. These code snippets usually follow a specific signature so that a worker can identify it and invoke it. In Listing 3.1, we show an example of a simple OpenWhisk function written in JavaScript. OpenWhisk functions expect a "main" procedure with a JSON input parameter, and another JSON as output. The "body" field of the output JSON is also treated as the HTML body of the response when the function is invoked via HTTP.

```
1  function main(args) {
2    const name = args.name || 'World';
3    return {
4      body: '<h1>Hello, ' + name + '!</h1>',
5    };
6  }
```

Listing 3.1: Example OpenWhisk Javascript function.

Controller

This component is the heart of a FaaS platform, acting as both the orchestrator for function execution and the provider of the service to the user. It usually consists of a set of APIs to create, read, update, and delete functions, and possibly other resources like "packages" or "modules" that can be used to group functions together, or trigger rules that can be used to link functions to events. The Controller is also responsible for function scheduling, so when a function is invoked, it decides which Worker should run it. It can also keep track of the state of the platform and the health of the Workers to inform the scheduling decisions. Furthermore, a growing number of platforms now also offer function

composition, allowing multiple functions to be combined into a single workflow. A popular form of composition on FaaS platforms is function chaining, where one function's output directly feeds into the next. The controller usually manages the chaining of functions, coordinating the invocation of the next function in the chain with the output of the previous one. This chaining allows for complex processes to be built from simpler functions.

Workers

The workers, also commonly called "invokers", are the function executors. They are typically distributed across a cluster of nodes and are accessible by the controller to request invocations. When such requests are received, the worker will prepare the execution environment, run the function, and return the result.

Runtimes The technologies that Workers employ to make this happen are often called "runtimes". The most common approach to implement runtimes are Containers [19], but new approaches have found use in recent years such as MicroVMs [118] (e.g. AWS Lambda moved to MicroVMs for function execution with FireCracker [3]), UniKernels [69], and WebAssembly [103]. The way these technologies are used is similar, as the objective is to have an isolated environment that can be quickly instantiated (and removed) with all the necessary dependencies to run functions.

In the early days of FaaS, platforms also used virtual machines to reach a maximum degree of isolation between invocations [94], but the overhead of instantiating and setting up VMs limited scaling capabilities. At that point, other approaches were explored, namely containers and microVMs. A container consists of a lightweight, isolated environment that packages applications with their dependencies, sharing the host OS kernel. On the other hand, microVMs are minimal virtual machines that include an OS kernel, keeping a similar level of isolation as VMs but with a lower overhead, similar to containers. Unikernels [69] are highly specialized, single-purpose machine images that include only the necessary parts of an operating system together with the application code and run directly on a hypervisor or on bare metal. Finally, WebAssembly [103] (commonly called Wasm) is a binary format compilation target for languages such as C/C++, Rust, and Go. It enables execution of code written in these languages within Web browsers with near-native speed. After the introduction of the WebAssembly System Interface (WASI), it can also be used to run standalone

programs through external calls (imports) to interact with a host environment, enabling the usage of Wasm modules in server-side environments and as FaaS runtimes.

Function Distribution Besides the way Workers perform function executions, they can follow different modalities on how they manage function instances. There are two main approaches to function distribution:

- **Single Function Worker**: a Worker instance is created for each function deployment. Here, the Worker is a thin wrapper around the runtime. When an user uploads a function, the platform deploys a new Worker instance with the corresponding runtime and function code, ready to accept invocation requests. The controller can then schedule to one of the available workers designated to run the function.
- **Multi-Function Worker**: a single Worker instance can instantiate multiple runtimes. This approach is also shown in Figure 3.2, where a Worker manages multiple function instances. With container-based runtimes, the Worker will manage a pool of containers and instantiate a new one when the controller requests an invocation for a specific function for the first time. It can then reuse the container for future invocations of the same function, to avoid the overhead of spinning up a new container. With WebAssembly runtimes, the Worker can just cache the compiled Wasm module and instantiate a new instance of the module for each invocation.

Storage

Intuitively, a storage component is crucial for managing platform artifacts, including function code, logs, and metadata. Multiple storage systems can be employed to store the different types of data. Invocations requests might also be stored for monitoring, to support workflows and retries, and for billing purposes.

Triggers

An equally-important aspect in a FaaS platforms are the triggering events that produce invocation requests. A trigger is a logical entity responsible for detecting or receiving external information and transforming it into internal events that initiate function execution. They are typically mapped to one or more functions in order to invoke

them via requests from a variety of heterogeneous sources. Depending on the platfom, multiple protocols (e.g., HTTP, WebSockets or messaging queues) and data formats (e.g., JSON, XML or binary) can be used. Commonly, FaaS platforms provide a set of built-in triggers, in particular HTTP requests and timers, to which users can add external services such as message queues via a trigger, connected to specific functions. The lifecycle and management of triggers are fully handled by the platform.

3.1.2 Example of FaaS Platform: Apache OpenWhisk

Apache OpenWhisk's architecture closely resembles the one depicted in Figure 3.2 with one, or more, Controllers and multi-function Workers (they spawn one container per function). It is an open-source FaaS platform initially developed by IBM and donated to the Apache Software Foundation. We report in Figure 3.3 a scheme of the architecture of OpenWhisk.

From left to right, we first find *Nginx*, which acts as the gateway and load balancer to distribute the incoming requests. *Nginx* forwards each request to one of the *Controller*s in the current deployment.

The *Controller*s decide on which of the available computation nodes, called *Worker*s[1], to schedule the execution of a given function. *Controller*s and *Worker*s do not interact directly but use Apache *Kafka* [62] and *CouchDB* [7] to respectively handle the routing and queueing of execution requests and to manage the authorisations and the storage of functions and of their outputs/responses.

*Worker*s execute functions using Docker containers. To schedule executions, *Controller*s follow a hard-coded policy that mediates load balancing and caching. This logic works by trying to allocate requests to the same functions on the same *Worker*s, hence saving time by skipping the retrieval of the function from *CouchDB* and the instantiation of the container already cached in the memory of the *Worker*.

In OpenWhisk, Workers follow the Multi-Function Worker approach, where a single Worker instance can instantiate multiple runtimes (using containers).

[1] OpenWhisk's documentation uses the more specific term "invokers".

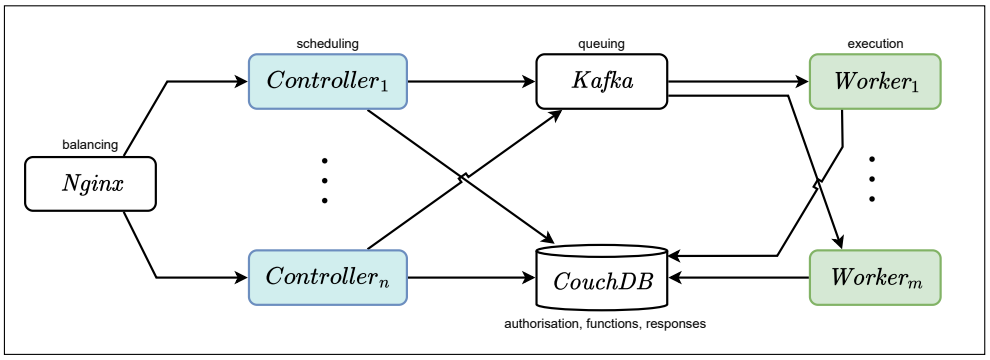


Figure 3.3: Apache OpenWhisk architecture.

3.2 Functions Scheduling

The Serverless development cycle is divided in two main parts: a) the writing of a function using a programming language supported by the platform (e.g., JavaScript, Python, C#) and b) the definition of an event that should trigger the execution of the function. For example, an event is a request to store some data, which triggers a process managing the selection, instantiation, scaling, deployment, fault tolerance, monitoring, and logging of the functions linked to that event. A Serverless provider is responsible to schedule functions on its workers, to control the scaling of the infrastructure by adjusting their available resources, and to bill its users on a per-execution basis.

3.2.1 Limitations

When instantiating a function, the provider has to create the appropriate execution environment for the function, as discussed in Section 3.1.1. How the provider implements the allocation of resources and the instantiation of execution environments impacts on the performance of the function execution [54, 13, 47, 46].

One of the main challenges to address is how should Serverless providers schedule the functions on the available computation nodes. To visualize the problem, consider, for example, Figure 3.4 which depicts the availability of two groups of heterogeneous Workers and a Controller. One group contains nodes with different computational resources, in particular one node is equipped with a GPU. The other group provides a Data Storage service and a Worker that is close to it.

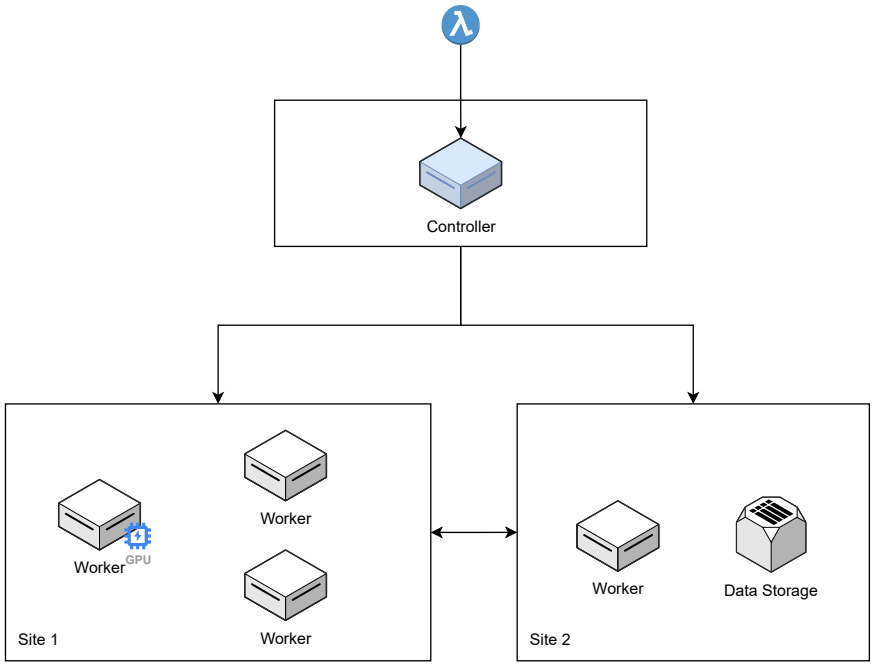


Figure 3.4: Scenario depicting multiple zones with heterogeneous workers. A naive scheduling algorithm would assign functions to workers without considering the kind of computational resources available.

All the Workers can execute a function that interacts with the Data storage. When the Controller (acting as the function scheduler) receives a request to execute the function, it must decide on which Worker to execute it. To minimise the response time, the Controller should consider the different computational loads of the Workers, which influence the time they take to execute the function. Moreover, the latency to access the Data storage plays an important role in determining the performance of function execution. The Worker at Site 2 is close to the data storage and enjoys a faster interaction with it while the Workers at Site 1 are farther from it and can undergo heavier latencies.

Cold Starts One of the most common slow-down issues in FaaS platforms is the "cold start" problem, where the first invocation of a function is slower than subsequent ones. If the provider allocates, for example, a new container for every request, the initialisation overhead of the container would negatively affect both the performance of the single function and heavily increase the load on the worker. The "cold start" is due

to the time it takes for the worker to spin up the new container with the appropriate dependencies for the function, initialize it with the function code, and then finally launch the execution. The opposite of a cold start would be a "warm start", where the worker can reuse a container that has already been initialized and is ready to run the function. A solution to tackle this problem is to maintain a "warm" pool of already-allocated containers. This principle is usually referred to as code locality [47].

Localities Resource allocation also includes I/O operations that need to be properly considered. For example, Wang et al. [115] report that a single function in the Amazon serverless platform can achieve on average 538Mbps network bandwidth, an order of magnitude slower than single modern hard drives (the authors report similar results from Google and Azure). Those performance result from bad allocations over I/O-bound devices, which can be reduced following the principle of session locality [47], i.e., taking advantage of already established user connections to workers. Another important aspect to consider to schedule functions, as underlined by the example in Figure 3.4, is that of data locality, which comes into play when functions need to intensively access (connection- or payload-wise) some data storage (e.g., databases or message queues). Intuitively, a function that needs to access some data storage and that runs on a worker with high-latency access to that storage (e.g., due to physical distance or thin bandwidth) is more likely to undergo heavier latencies than if run on a worker "closer" to it. Data locality has been subject of research in neighbouring Cloud contexts [124, 117].

3.2.2 Scheduling in Apache Openwhisk

Given a function to be executed, OpenWhisk's controller acts as the load balancer by forwarding the execution request to one selected worker.

The load balancing policy followed by the controller aims at maximising container reuse. When the controller needs to schedule the execution of a function, a numeric hash h is calculated using the action name. A worker is then selected using the remainder of the division between h and the total number of workers n. The controller checks if the worker is overloaded. If the chosen worker is overloaded, the index is incremented by a step-size, which is any of the co-prime numbers smaller than the amount n of available workers.

When no worker is available after cycling through the entire worker pool, the load balancer randomly selects a worker from those that are considered "healthy"—able to sustain the workload. This happens when there are workers that are healthy but have no capacity available when the scheduling algorithm was searching for a worker. When there are no healthy workers, the load balancer returns an error stating that no workers are available for executing the function.

Part II

Contributions

Chapter 4

Allocation Priority Policies

4.1 Introduction

The first challenge we address in this thesis, is the problem of *function-execution scheduling optimisation* [47], as discussed in Section 3.2.1, for which we propose a methodology that enables Serverless providers to efficiently schedule functions on available computation nodes using a declarative language called *Allocation Priority Policies* (APP). Developers can use APP to specify a scheduling policy for their functions that the scheduler later uses to find the worker that, given the current status of the system, best fits the constraints specified by the developer of a given function. To substantiate our proposal, we extended the scheduler of OpenWhisk to use APP-defined policies in the scheduling of Serverless functions.

Structure of the chapter In Section 4.2 we detail the APP language and present our prototypical implementation as an extension of OpenWhisk. To validate our extension, in Section 4.3, we present a use case combining IoT, Edge, and Cloud Computing and in Section 4.4 we contrast an implementation of the use case using our APP-based prototype with a naïve one using three coexisting installations of the vanilla OpenWhisk stack to achieve the same functional requirements. We present the data on the performance of the two deployments, providing empirical evidence of the performance gains offered by the APP-governed scheduling. We discuss future and concluding remarks in Section 4.5.

4.2 The APP Language

At least three aspects related to function scheduling affect the performances of function execution in Serverless platforms: code, session, and data locality. Load balancing policies adopted by state-of-the-art Serverless platforms like Apache OpenWhisk take advantage only of code locality, but they currently have no way to integrate also information on other types of locality. To take advantage of other forms of locality, the load balancer should have knowledge on the way functions access external resources, like I/O-bound devices or databases, whose usage depends on the implementation of functions. As a first work to tackle this issue, we aim at bridging that information gap, presenting a language that any Serverless platform can use in its scheduling policies to consider those factors. Our approach is conservative: with its default settings (explained in the next section) it can capture the status of current Serverless platforms. Then, more advanced Serverless users and platform providers can use the features offered by our proposal to optimise the execution of functions. Moreover, optimised scheduling policies could be the outcome of automatic heuristic/inference systems applied to the functions to be executed. With this chapter we address the first fundamental step, i.e., showing the feasibility of Serverless platforms instructed with customized load balancing rules. As previously discussed, current serverless platforms, like OpenWhisk, come equipped with hard-coded load balancing policies. In this section, we present the *Allocation Priority Policies* (APP) language, intended as a language to specify customised load balancing policies and overcome the inflexibility of the hard-coded load balancing ones. The idea is that both developers and providers can write, besides the functions to be executed by the platform, a policy that instructs the platform what workers each function should be preferably executed on. Function-specific configurations are optional and without them the system can follow a default strategy.

As an extension of the example depicted in Figure 3.4, consider some functions that need to access a database. To reduce latency (as per data locality principle), the best option would be to run those functions on the same pool of machines that run the database. If that option is not valid, then running those functions on workers in the proximity (e.g., in the same network domain) is preferable to using workers located further away (e.g., in other networks). Below, Listing 4.1, we provide an initial APP script that specifies the scheduling policies only for those workers belonging to the pool

$$
\begin{array}{lcl}
\textit{policy\_tag} & \in & \textit{Identifiers} \cup \{\texttt{default}\} \qquad \textit{worker\_label} \in \textit{Identifiers} \qquad n \in \mathbb{N} \\
\textit{app} & ::= & \overline{\textit{tag}} \\
\textit{tag} & ::= & \textit{policy\_tag} : \overline{\texttt{-}\ \textit{block}}\ \textit{followup}? \\
\textit{block} & ::= & \texttt{workers}\ [\ \texttt{"*"} \mid \overline{\texttt{-}\ \textit{worker\_label}}\ ] \\
 & & (\texttt{strategy}\ [\ \texttt{random} \mid \texttt{platform} \mid \texttt{best\_first}\ ])? \\
 & & (\texttt{invalidate}\ [\ \texttt{capacity\_used} : n\texttt{\%} \\
 & & \quad \mid \texttt{max\_concurrent\_invocations} : n \mid \texttt{overload}\ ])? \\
\textit{followup} & ::= & \texttt{followup} : [\ \texttt{default} \mid \texttt{fail}\ ]
\end{array}
$$

Figure 4.1: The APP syntax.

of machines running the database.

```
couchdb_query:
  - workers:
    - DB_worker1
    - DB_worker2
  strategy: random
  invalidate:
    capacity_used: 50%
  followup: fail
```

Listing 4.1: Simple APP script for data locality optimization.

At the first line, we define the *policy tag*, which is `couchdb_query`. As explained below, tags are used to link policies to functions. Then, the keyword `workers` indicates a list of *worker labels*, which identify the workers in the proximity of the database, i.e., `DB_worker1` and `DB_worker2`. As explained below, labels are used to identify workers. Finally, we define three parameters: the `strategy` used by the scheduler to choose among the listed worker labels, the policy that `invalidate`s the selection of a worker label, and the `followup` policy in case all workers are `invalidate`d. In the example, we select one of the two labels `random`ly, we `invalidate` their usage if the workers corresponding to the chosen label are used at more than the `50%` of their capacity (`capacity_used`) and, in case all workers are invalidated (`followup`), we let the request for function execution `fail`.

The APP syntax and semantics We report the syntax of APP in Figure 4.1. The basic entities considered in the APP language are *a*) scheduling policies, identified by a *policy tag* identifier to which users can associate their functions—the policy-function

association is a one-to-many relation—and *b*) workers, identified by a *worker label*—where a label identifies a collection of computation nodes. An APP script is a YAML [18] file specifying a sequence of policies. Given a tag, the corresponding policy includes a list of workers blocks, possibly closed with a followup strategy. A workers block includes three parameters: a collection of worker labels, a possible scheduling strategy, and an invalidate condition. A followup strategy can be either a default policy or the notification of failure.

We discuss the APP semantics, and the possible parameters, by commenting on a more elaborate script extending the previous one, shown in Listing 4.2. The APP script starts with the tag `default`, which is a special tag used to specify the policy for non-tagged functions, or to be adopted when a tagged policy has all its members invalidated, and the followup option is default.

In Listing 4.2, the `default` tag describes the default behaviour of the serverless platform running APP. The wildcard "*" for the workers represent all worker labels. The strategy selected is the platform default (e.g., in our prototype in Section 4.3 the platform strategy corresponds to the selection algorithm described in Section 3.2.2) and its invalidate strategy considers a worker label non-usable when its workers are overloaded, i.e., none has enough resources to run the function.

```
1  default:
2    - workers: "*"
3      strategy: platform
4      invalidate: overload
5
6  couchdb_query:
7    - workers:
8      - DB_worker1
9      - DB_worker2
10     strategy: random
11     invalidate: capacity_used: 50%
12   - workers:
13     - near_DB_worker1
14     - near_DB_worker2
```

```
15        strategy: best_first
16        invalidate: max_concurrent_invocations: 100
17      followup: fail
```

Listing 4.2: Example of an APP script.

Besides the `default` tag, the `couchdb_query` tag is used for those functions that access the database. The scheduler considers worker blocks in order of appearance from top to bottom. As mentioned above, in the first block (associated to `DB_worker1` and `DB_worker2`) the scheduler randomly picks one of the two worker labels and considers a label invalid when all corresponding workers reached the 50% of capacity. Here the notion of capacity depends on the implementation (e.g., our OpenWhisk-based APP implementation in Section 4.3 uses information on the CPU usage to determine the load of invokers). When both worker labels are invalid, the scheduler goes to the next workers block, with `near_DB_worker1` and `near_DB_worker2`, chosen following a best_first strategy—where the scheduler considers the ordering of the list of workers, sending invocations to the first until it becomes invalid, to then pass to the next ones in order. The invalidate strategy of the block regards the maximal number of concurrent invocations over the labelled workers—max_concurrent_invocations, which is set to 100. If all the worker labels are invalid, the scheduler applies the followup behaviour, which is to fail.

Summarising, given a policy tag, the scheduler considers the corresponding workers blocks starting from the top. A block includes three parameters:

- `workers`: contains a non-empty list of worker labels or the `"*"` wildcard to encompass all of them;
- `strategy`: defines the policy of worker label selection. APP currently supports three strategies:
  - `random`: labels are selected in a fair random manner;
  - `best_first`: labels are selected following their order of appearance;
  - `platform`: labels are selected following the default strategy of the serverless platform—in our prototype (cf. Section 4.3) the `platform` option corre-

sponds to the algorithm based on identifier hashing with co-prime increments explained in Section 3.2.2.

- `invalidate`: specifies when to stop considering a worker label. All invalidate options below include as preliminary condition the unreachability of the corresponding workers. When all labels in a block are invalid, the next block or the followup behaviour is used. Current `invalidate` options are:
  - `overload`: the corresponding workers lack enough computational resources to run the function;[1]
  - `capacity_used`: the corresponding workers reached a threshold percentage of CPU load (although not being overloaded);
  - `max_concurrent_invocations`: the corresponding workers have reached a threshold number of buffered concurrent invocations.
- `followup`: specifies the policy applied when all the blocks in a policy tag are considered invalid. The supported followup strategies are:
  - `fail`: stop the scheduling of the function;
  - `default`: follow what is defined in the `default` tag.

4.3 APP Implementation in Apache OpenWhisk

We have implemented a serverless platform in which load balancing policies can be customised using the APP language. This implementation[2] was obtained by modifying the OpenWhisk code base. Namely, we have replaced the load balancer module in the OpenWhisk controller, with a new one that reads an APP script, parses it, and follows the specified load balancing policies when OpenWhisk invokers should be selected[3].

[1]The kind of computational resources that determine the `overload` option depends on the APIs provided by a given serverless platform. For example, in our prototype in Section 4.3 we consider a worker label `overload`ed when the related invokers are declared "unhealthy" by the OpenWhisk APIs, which use memory consumption and CPU load.

[2]The implementation is on an open-source a fork of Apache OpenWhisk. Available at `https://github.com/giusdp/openwhisk`.

[3]For simplicity, we chose to associate one worker label with one invoker.

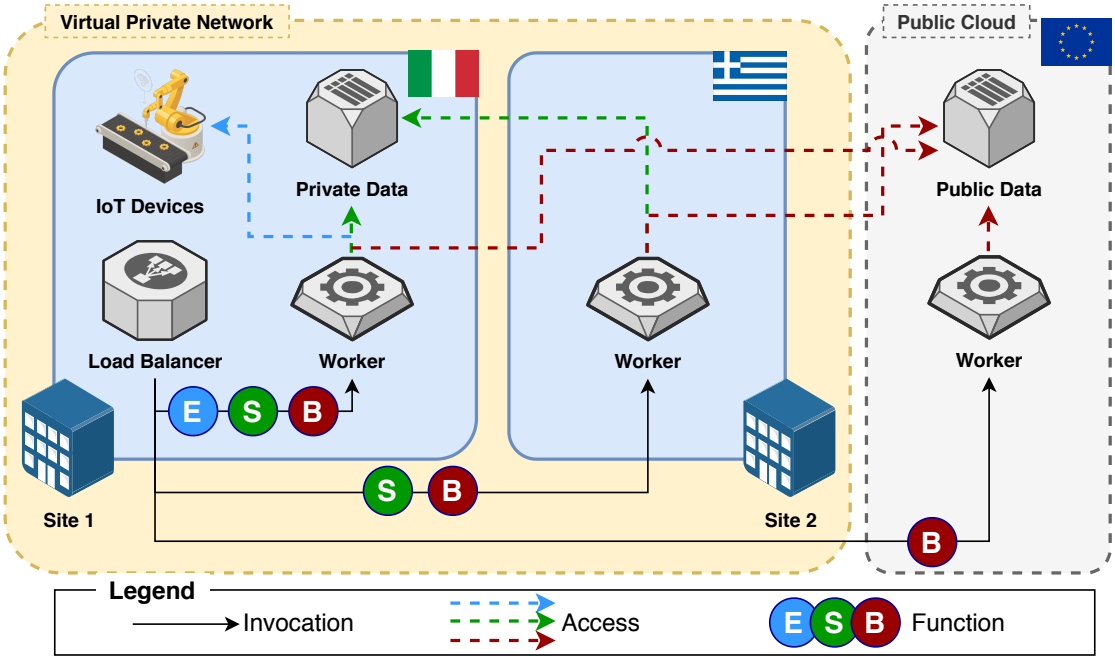


Figure 4.2: Use case architecture representation.

To test our implementation, we used the Serverless use case depicted in Figure 4.2 encompassing three Serverless domains: *i*) a private cloud with a low-power edge-device **Worker** at a first location, called **Site 1**; *ii*) a private cloud with the **Worker** at **Site 1** and a mid-tier server **Worker** at a second location, called **Site 2**; *iii*) a hybrid cloud with the two **Workers** at **Site 1** and **Site 2** and a third mid-tier server from a **Public Cloud**. **Site 1** and **Site 2** are respectively located in Italy and Greece while the **Public Cloud** is located in northern Europe.

Site 1 is the main branch of a company and it runs both a data storage of **Private Data** and the **IoT Devices** used in their local line of production. **Site 1** also hosts the scheduler of functions, called the **Load Balancer**. The **Worker** at **Site 1** can access all resources within its site. **Site 2** hosts a **Worker** which, belonging to the company virtual private network (VPN), can access the **Private Data** at **Site 1**. The company also controls a **Worker** in a **Public Cloud** and a data storage with **Public Data** accessible by all **Workers**.

In the use case, three different function deployments need to co-exist in the same infrastructure, marked as E , S , and B . Function E (edge) manages the **IoT Devices** at **Site 1** and it can only execute on the edge **Worker** at the same location,

which has access to those devices. Function S (small) is a light-weight computation that accesses the **Private Data** storage at **Site 1**, within the company VPN. Function B (big) performs heavy-load queries on the **Public Data** storage in the **Public Cloud**. As mentioned, here data locality plays an important part in determining the performance of Serverless function execution:

- the **Worker** at **Site 1** can execute all functions. It is the only worker that can execute E and it is the worker with the fastest access to the co-located **Private Data** for S . It can execute B undergoing some latency due to the physical distance with the **Public Data** storage;
- the **Worker** at **Site 2** can execute functions S and B , undergoing some latency on both functions due to its distance from both data storages;
- the **Worker** at the **Public Cloud** can execute B , enjoying the fastest access to the related **Public Data** source.

4.4 Experiments and Results

We compare the differences on the architecture and performance of the use case above as implemented using our APP-based OpenWhisk prototype against a naïve implementation using the vanilla OpenWhisk.

Specifically, we implement the use case using a Kubernetes cluster composed of a low-power device—with an Intel Core i7-4510U CPU with 8GB of RAM—in Italy for **Site 1**, a Virtual Machine—comparable to an Amazon EC2 a1.large instance—from the Okeanos Cloud (`https://okeanos.grnet.gr`) located in Greece for **Site 2**, and a Virtual Machine—comparable to an Amazon EC2 a1.large instance—from the **Public Cloud** of Microsoft Azure located in Northern Europe.

Following the requirements of the use case, we define the APP deployment plan for the use case as follows:

```
Function_E:
  - workers:
    - worker_site1
  followup: fail
```

```
Function_S:
  - workers:
    - worker_site2
    - worker_site1
    strategy: random
  followup: fail
```

```
Function_B:
  - workers:
    - worker_public_cloud
    - worker_site2
    - worker_site1
    strategy: best_first
  followup: fail
```

Commenting the code above, we have function (E) represented by `Function_E`, where the only invoker available is the one at **Site 1** (`worker_site1`). Since we do not allow other invokers to handle (E), we set the `followup` value to `fail`. For (S) we have `Function_S`, where the invokers available are the ones at **Site 1** and **Site 2** (`worker_site2`). We let the two invokers split evenly the load of invocations, assigning `random` as routing `strategy`. Also here we let the invocation `fail` since we do not have other invokers able to access the **Private Data** storage within the company VPN. Finally, the policy for (B) (`Funcion_B`) includes all workers (hence also `worker_public_cloud` besides the ones at **Site 1** and **Site 2**) selected according to the `best_first` strategy. As for (S), also here we let the invocation `fail` since no other invokers are available.

For the APP-based deployment, we locate the Load Balancer at **Site 1** registering to it the three **Worker**s/invokers from **Site 1**, **Site 2** and the **Public Cloud**. For the naïve implementation, we use the same cluster but we install three separate but co-existing

vanilla OpenWhisk instances. The three separate instances are needed to implement the functional requirements of limiting the execution of function E only on the Italian **Worker**, of S only on the Italian and Greek **Workers**, and of B on all **Worker**s.

To implement the databases (both **Private** and **Public** ones) we used a CouchDB instance deployed at **Site 1** and another in the **Public Cloud**. To simulate the access to IoT devices at **Site 1** (function E) we implemented a JavaScript function that, queried, returns some readings after a one-second delay. We followed a similar strategy for S and B , where two JavaScript functions perform a (respectively lighter and heavier) query for JSON documents.

Architectural Evaluation An evident problem that arises with the triple-deployment combination is the increased consumption of computational and memory resources to host 3 copies of all the components, most importantly the Controller and the Invoker. A partial solution to this is to deploy separately the Kafka, Redis, and CouchDB components used by OpenWhisk, configuring them to be used by the three different installations simultaneously. However, we did not perform such optimisation to minimise the differences between the two tested architectures.

Quantitative Evaluation To have statistically relevant figures to compare the two setups (the APP-based and the vanilla one), we fired a sequence of 1000 requests for each function in each setup. We report the results of the tests of the APP-based implementation in Table 4.1 and those of the vanilla one in Table 4.2. In both tables, the first column on the left reports the tested function. The three following columns report the number of requests served by the respective **Worker**s at **Site 1**, **Site 2**, and in the **Public Cloud**. The last two columns report the time passed from sending a request to the reception of its response: the second-to-last column reports the average time (in ms) and the last one reports the average time (in ms) for the fastest 95^{th} percentile of request-responses.

We comment on the results starting from E (first row from the header in both tables). As expected, all requests for E are executed at **Site 1**. The slight difference in the two averages (APP ca. 5.6% faster than vanilla) and the two fastest 95^{th} percentile (APP ca. 0.6% faster than vanilla) come from the heavier resource consumption of the vanilla deployment.

	Site 1	Site 2	Public Cloud	Average (ms)	95% Average (ms)
E	1000	0	0	1096.53	1019.03
S	466	534	0	149.18	90.86
B	0	90	910	105.18	64.62

Table 4.1: 1000 invocation for each function in the APP-based OpenWhisk deployment.

	Site 1	Site 2	Public Cloud	Average (ms)	95% Average (ms)
E	1000	0	0	1159.90	1025.52
S	19	981	0	385.30	302.08
B	185	815	0	265.69	215.793

Table 4.2: 1000 invocations for each function in the vanilla OpenWhisk deployment.

As expected, the impact of data locality and the performance increase provided by the data-locality-aware policies in APP become visible for S and B . In the case of S , the Load Balancer of the vanilla deployment elected **Site 2** as the location of the main invoker (passing to it 98.1% of the invocations). We remind that S accesses a **Private Data** storage located at **Site 1**. The impact of data locality is visible on the execution of S in the vanilla deployment, being 88.35% slower than the APP-based deployment on average and 107.5% slower for the fastest 95th percentile. On the contrary, the APP-based scheduler evenly divided the invocations between **Site 1** (46.6%) and **Site 2** (53.4%) with a slight preference for the latter, thanks to its greater availability of resources. In the case of B , the Load Balancer of the vanilla deployment elected again **Site 2** as the location of the main invoker (passing to it 81.5% of all the invocations) and **Site 1** as the second-best (passing the remaining 18.5%). Although available to handle computations, the invoker in the **Public Cloud** is never used as the other two managed to handle the load. Since B accesses a **Public Data** storage located in the **Public Cloud**, also in this case the effect of data locality is strikingly visible, marking a heavy toll on the execution of B in the vanilla deployment, which is 86.5% slower than the APP-based deployment on average and 107.8% slower for the fastest 95th percentile. The APP-based scheduler, following the preference on the **Public Cloud**, sends the majority of invocations to the **Public Cloud** (91%) while the invocations that exceed the resource limits of the **Worker** in the **Public Cloud** are routed to **Site 2** (9%), as defined by the `Function_E` policy.

As a concluding remark over our experiment, we note that these results do not

prove that the vanilla implementation of OpenWhisk is generally worse (performance-wise) than the APP-based one. Indeed, what emerged from the experiment is the expected result that, without proper information and software infrastructure to guide the scheduling of functions with respect to some optimisation policies, the Load Balancer of OpenWhisk can perform a suboptimal scheduling of function executions. Hence, there was a chance that the Load Balance of OpenWhisk could have performed some better scheduling strategies in our experiment, however that would have been an occasional occurrence rather than an informed decision. Contrarily, when equipped with the proper information (as it happens with our APP-based prototype) the Load Balancer can reach consistent results, which is the base for execution optimisation.

4.5 Conclusions

We started addressing the problem of *function-execution scheduling optimisation* by proposing a methodology that provides developers with a declarative language called APP to express scheduling policies for functions, laying the foundation for a family of frameworks that can be used to enhance new and existing FaaS platforms. To validate our work, we extended the scheduler of OpenWhisk to use APP-defined policies in the scheduling of Serverless functions and empirically tested our extension on a use case that combines IoT, Edge, and Cloud Computing, contrasting our implementation with a naïve one using the vanilla OpenWhisk stack to achieve the same functional requirements.

Chapter 5

Topology-aware Serverless Scheduling

5.1 Introduction

With APP we introduced a novel way to specify customised load-balancing policies so that developers can optimize their serverless applications by exploiting the locality of the resources. Use-cases like the one in Figure 5.1 motivate the need for such policies.

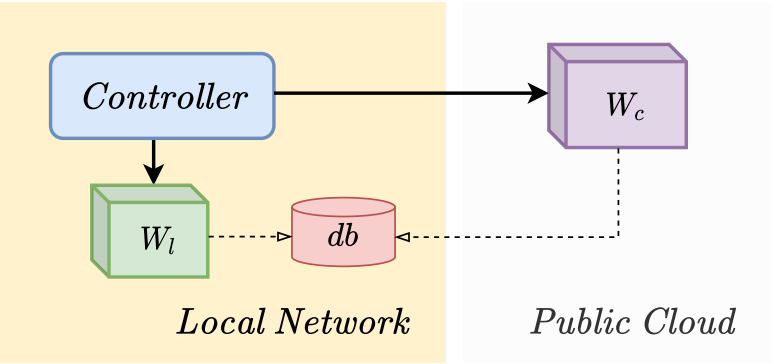


Figure 5.1: Example of function-execution scheduling problem.

We have a simple serverless system composed of two workers. One worker, W_l, executes in the local network and the other, W_c, is in a public cloud. Both workers can execute functions that interact (represented by the dashed lines) with a database *db* deployed in the local network. When the *Controller* (acting as function scheduler)

receives a request to execute the function, it must decide on which worker to execute it. To minimise the response time, the *Controller* should consider the different computational power of the workers as well as their current loads, which influence the time they take to execute the function. Moreover, the performance of the functions that interact with the database depends on the latter's access latency of the node they run on: W_l is close to the *db* and enjoys a faster interaction with it while W_c is farther away and can undergo heavier latencies.

With such use-cases, APP is enough to fine-tune serverless applications to improve resource usage. However, in more complex scenarios where the system comprises of replicated controllers and many workers distributed and isolated in different zones, the language fails to capture the system's complexity. For this reason we developed an extension of APP.

A Motivating Example We further clarify the concepts of locality-bound FaaS scheduling with a case study from our industry partners, which we use as an example throughout the chapter. We deem the case useful to help understand our contribution and clarify the motivation behind our work.

The case concerns a cloud-edge-continuum system to control and perform both predictive maintenance and anomaly detection over a fleet of robots in a production line. The system runs three kinds of computational tasks: i) predictions of critical events, performed by analysing data produced by the robots, ii) non-critical predictions and generic control activities, and iii) machine learning tasks. Tasks i) follow a closed-control loop between the fleet that generates data and issues these tasks and the workers that run them and can act on the fleet. Since tasks i) can avert potential risks, they must execute with the lowest latency and their control signals must reach the fleet urgently. The users of the system launch the other kinds of tasks, which have no time-constrains. Tasks iii) have resource-heavy requirements. We depict the solution that we have designed for the deployment of the system in Figure 5.2. We consider three kinds of functions, one for each kind of tasks: critical functions ⓘ, generic functions ◇, and machine learning functions ☁. To guarantee low-latency and the possibility to immediately act on the robots, we execute *critical* functions ⓘ on edge devices (workers $W_1, ..., W_i$ in Figure 5.2) directly connected to the robots. Since machine-learning algorithms require considerable resources, which the company prefers to provision

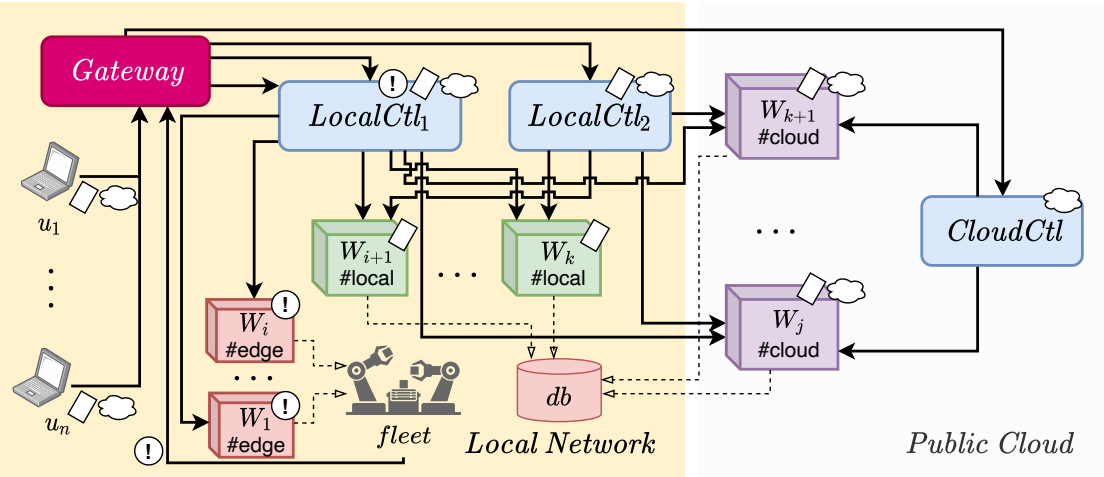


Figure 5.2: Representation of the case study.

on-demand, we execute the machine-learning functions ☁ on a public cloud, outside the company's perimeter (W_{k+1},...,W_j in Figure 5.2). The generic functions ◇ do not have specific, resource-heavy requirements, but they might need to access the database *db* in the local network. Hence, we schedule these preferably on the local cluster (W_{i+1},...,W_k in Figure 5.2) and use on-demand public-cloud workers when the local ones are at full capacity.

For performance and reliability, our solution considers two function-scheduling controllers for the internal workers, i.e., the controllers $LocalCtl_1$ and $LocalCtl_2$, and one for cloud workers, i.e., the controller *CloudCtl*. One local controller, namely $LocalCtl_1$, has a dedicated low-latency connection with the edge devices able to act on the fleet.

Finally, a *Gateway* acts as load balancer among the controllers. However, to follow the requirements of the company, instead of adopting a generic round-robin policy, we need to instruct the *Gateway* to forward critical functions ⓘ to $LocalCtl_1$, the generic functions ◇ to one between $LocalCtl_1$ and $LocalCtl_2$, and the cloud functions ☁ to *CloudCtl* (or to any other controller when the latter is not available).

The APP Extension The case above presents a scenario where we need to deploy the serverless platform over at least two zones (local network and public cloud) and where the function-execution scheduling policy depends on a topology of different clusters (edge-devices, local cluster, and cloud cluster). The scheduling policies influence the

behaviour of both the gateway and the controllers, which need to know the current status of the workers (e.g., to execute generic functions in the cloud when the local cluster is overloaded). One can obtain a deployment of the case by modifying the source code of all the involved components and by hard-coding their desired behaviour. However, this solution requires a deep knowledge of the internals of the components and is fragile and difficult to maintain.

We propose an approach based on a new declarative language, called tAPP (Topology-aware Allocation Priority Policies), used to write *configuration* files describing topology-aware function-execution scheduling policies. In this way, following the Infrastructure-as-Code philosophy, users (typically DevOps) can keep all relevant scheduling information in a single repository (in one or more tAPP files) which they can version, change, and run without incurring downtimes due to system restarts to load new configurations.

Structure of the chapter We first introduce the tAPP language with an example in Section 5.2 and detail its syntax in Section 5.3. We implement support for tAPP in OpenWhisk, which allows us to evaluate the feasibility of topology-aware scheduling policies, presented in Section 5.4. We show that our prototype can capture typical functional scheduling requirements in cloud-edge deployments that cannot be supported by standard deployments of vanilla OpenWhisk. We detail the impact of tAPP on the locality-bound scenario described in our motivating example in Section 5.4.1. In Section 5.4.2, we analyse the overhead of the tAPP-based extension of OpenWhisk w.r.t. the vanilla version through test cases drawn from *ServerlessBench* [128], a benchmark suite for serverless platforms. In Section 5.5 we draw conclusions discussing future work.

5.2 tAPP, by example

The Topology-aware Allocation Priority Policy (tAPP) language is a declarative language able to specify customised load-balancing policies and overcome the inflexibility of the hard-coded load-balancing ones. The idea is that tAPP can support developers and providers in optimising the execution of serverless functions. tAPP is tailored to adapt to the different types of information on the serverless infrastructure that providers share with developers. For example, in edge deployments (where it is important to

know on which machine functions run), developers know which nodes are available and their position; in the Cloud, developers think in terms of regions (e.g., west/east US, Europe) and zones (Los Angeles, New York, Paris areas), rather than single nodes. As exemplified in Section 5.1, tAPP policies can scale according to these needs and adapt to cloud-edge-continuum scenarios, where policies can span single nodes, unbound collections of these (e.g., defined by some common trait), and topological zones. tAPP can also work in the absence of information provided to developers—without function-specific configurations, tAPP-based platforms follow a default strategy, like the other, hardwired alternatives.

As an extension of the example depicted in Figure 5.1, consider some functions that need to access a database. To reduce latency (as per data locality principle), the best option would be to run those functions on the same pool of machines that run the database. If that option is not valid, then running those functions on workers in the proximity (e.g., in the same network domain) is preferable than using workers located further away (e.g., in other networks).

An initial tAPP script that specifies the scheduling policies only for those workers belonging to the pool of machines running the database can be the same as the original APP script showed in Listing 4.1. We can define a *policy tag*, associate some `workers` and optionally specify a `strategy`, an `invalidate` condition and the `followup` rule. Essentially, tAPP can capture the same policies as APP at its core, with the addition of new constructs to express potential topological constraints.

5.2.1 The tAPP Approach

A tag identifies a policy (e.g., we can use a tag "critical" to identify the scheduling behaviour of the critical ⓘ functions of our case study, cf. Section 5.1) and it marks all those functions that shall follow the same scheduling behaviour (e.g., marking as "critical" any function that falls into that category). Topologies are part of policies and come in two facets. *Physical* topologies relate to *zones*, which can represent availability zones in public clouds and plants in multi-plant industrial settings. *Logical* topologies instead represent partitions of workers. The logical layer expresses the constraints of the *user* and identifies the pool of workers which can execute a given function (e.g., for performance). The smallest logical topology is the singleton, i.e., a worker, which we

identify with a distinct label (e.g., W_1 in Figure 5.2). In general, policies can target lists of singletons as well as aggregate multiple workers in different sets. The interplay between the two topological layers determines which workers a controller can use to schedule a function. For example, we can capture the scheduling behaviour of the critical functions of our case study in this way: 1) we assign $LocalCtl_1$, $LocalCtl_2$, and W_1,...,W_k to the same zone, 2) we configure said workers to only accept requests from co-located controllers (this, e.g., excludes access to *CloudCtl*), and 3) we set the policy of the critical functions to only use the workers tagged with the *edge* label, #edge in Figure 5.2. Besides expressing topological constraints, policies can include other directions such as the strategy followed by the controller to choose a worker within the pool of the available ones (e.g., to balance the load evenly among them) and when workers are ineligible (e.g., due to their resource quotas).

5.3 The tAPP Language

policy_tag ∈ *Identifiers* ∪ {default} *label* ∈ *Identifiers* $n \in \mathbb{N}$

app	::=	- *tag*
tag	::=	*policy_tag* : - *controller?* *workers* *strategy?* *invalidate?* *strategy?* *followup?*
controller	::=	controller : *label* (topology_tolerance : (all — same — none))?
workers	::=	workers: - wrk : *label* *invalidate?*
	|	workers: - set : *label?* *strategy?* *invalidate?*
strategy	::=	strategy : (random | platform | best_first)
invalidate	::=	invalidate : (capacity_used n% | max_concurrent_invocations n | overload)
followup	::=	followup : (default | fail)

Figure 5.3: The syntax of tAPP (the extensions from APP are highlighted).

We report the syntax of tAPP in Figure 5.3.

tAPP scripts are YAML [18] files. The basic entities considered in the language are *a*) scheduling policies, defined by a *policy tag* identifier to which users can associate their functions—the policy-function association is a one-to-many relation—and *b*) workers, identified by a *worker label*—where a label identifies a collection of computation nodes. All identifiers are strings formed with the accepted character set as defined in [18].

Given a tag, the corresponding policy includes a list of blocks, possibly closed with `strategy` and `followup` options. A block includes four parameters: an optional

controller selector, a collection of workers, a possible scheduling strategy, and an invalidate condition. The outer strategy defines the policy we must follow to select among the blocks of the tag, while the inner strategy defines how to select workers from the items specified within a chosen workers block. The controller defines the identifier of a specific controller we want the gateway to redirect the invocation request to. When used, it is possible to define a topology_tolerance option to further refine how tAPP handles failures (of controllers). The collection of workers can be either a list of labels pointing to specific workers (wrk), or a worker set. In lists, the user can specify the invalidate condition of each single worker, while in sets, the invalidate condition applies to all the workers included in the set. When users specify an invalidate condition at block level, this is directly applied to all workers items (wrk and set) that do not define one. In sets the user can also specify a strategy followed to choose workers within the set. Finally, the followup value defines the behaviour to take in case no specified controller or worker in a tag is available to handle the invocation request.

We discuss the tAPP semantics, and the possible parameters, by commenting on a more elaborate script extending the previous one, shown in Listing 5.1. The tAPP script starts with the tag `default`, which is a special tag used to specify the policy for non-tagged functions, or to be adopted when a tagged policy has all its members invalidated, and the followup option is default.

In Listing 5.1, the `default` tag describes the default behaviour of the serverless platform running tAPP. In this case we use a workers set to select workers, with no value specified for set which represents all worker labels. The strategy selected is the platform default. In our prototype in Section 5.3.1 the platform strategy corresponds to a selection algorithm, discussed in Section 3.2.2, which mediates load balancing and code locality by associating a function to a numeric hash and a step-size. The invalidate strategy considers a worker non-usable when it is overloaded, i.e., it does not have enough resources to run the function.

```
1 - default:
2   - workers:
3     - set:
4       strategy: platform
```

```
5         invalidate: overload
6
7   - couchdb_query:
8     - workers:
9       - wrk: DB_worker1
10      - wrk: DB_worker2
11      strategy: random
12      invalidate: capacity_used 50%
13    - workers:
14      - wrk :near_DB_worker1
15      - wrk :near_DB_worker2
16      strategy: best_first
17      invalidate: max_concurrent_invocations 100
18    followup: fail
```

Listing 5.1: Example of a tAPP script.

Besides the `default` tag, the `couchdb_query` tag is used for those functions that access the database. The scheduler considers worker blocks in order of appearance from top to bottom. As mentioned above, in the first block (associated to `DB_worker1` and `DB_worker2`) the scheduler randomly picks one of the two worker labels and considers the corresponding worker invalid when it reaches the `50%` of capacity. Here the notion of capacity depends on the implementation (e.g., our OpenWhisk-based tAPP implementation uses information on the CPU usage to determine the load of invokers). When both worker labels are invalid, the scheduler goes to the next `workers` block, with `near_DB_worker1` and `near_DB_worker2`, chosen following a `best_first` strategy—where the scheduler considers the ordering of the list of `workers`, sending invocations to the first until it becomes invalid, to then pass to the next ones in order. The `invalidate` strategy of the block (applied to the single `wrk`) regards the maximal number of concurrent invocations over the labelled worker—`max_concurrent_invocations`, which is set to `100`. If all the worker labels are invalid, the scheduler applies the `followup` behaviour, which is to `fail`.

Users can define subsets of workers by specifying a label associated with the workers, e.g., `local` selects only those workers associated to the *local* label.

The scheduling on worker-sets follows the same logic of block-level worker selection: it exhausts all workers before deeming the item invalid. Since worker-set selection/invalidation policies are distinct from block-level ones, we let users define the `strategy` and `invalidate` policies to select the worker in the set. For example, we can pair the above selection with a *strategy* and an *invalidate* options, e.g.,

```
- workers:
  - set: local
    strategy: random
    invalidate: capacity_used 50%
```

which tells the scheduler to adopt the `random` selection strategy and adopt the `capacity_used` invalidation policy when selecting the workers in the *local* set. When worker-sets omit the definition of the selection `strategy` we consider the default one. When the invalidation option is omitted, we either use the one of the enclosing block or, if the latter is missing too, the default one.

Summarising, given a policy tag, the scheduler follows the policy defined in the `strategy` option to select the corresponding blocks. A block includes three parameters:

- `workers`: which either contains a non-empty list of worker (`wrk`) labels, each paired with an optional invalidation condition, or a worker-`set` label (possibly blank, to select all workers) to range over sets of workers; workers `set`s optionally define the `strategy` and `invalidate` options to select workers within the set and declare them invalid;
- `strategy`: defines the policy of item selection at the levels of *policy_tag*, *workers* block, and workers `set`s. APP currently supports three strategies:
  - `random`: selects items in a fair random manner;
  - `best_first`: selects items following their order of appearance;
  - `platform`: selects items following the default strategy of the serverless platform—in our prototype, this corresponds to a co-prime-based selection.
- `invalidate`: specifies when a worker (label) cannot host the execution of a function. All invalidate options include, as preliminary condition, the unreachability

of a worker. When all labels in a block are invalid, we follow the defined `strategy` to select the next block one until we either find a valid worker or we exhaust all blocks. In the latter case, we apply the `followup` behaviour. Current `invalidate` options are:

  - `overload`: the worker lack enough computational resources to run the function;[1]
  - `capacity_used`: the worker reached a threshold percentage of CPU load;
  - `max_concurrent_invocations`: the worker have reached a threshold number of buffered concurrent invocations.

- `followup`: specifies the policy applied when all the blocks in a policy tag are considered invalid. The supported follow up strategies are:

  - `fail`: drop the scheduling of the function;
  - `default`: apply the `default` tag.

Since the `default` block is the only possible "backup" tag used when all workers of a custom tag cannot execute a function (because they are all invalid), the `followup` value of the `default` tag is always set to `fail`.

Besides the above elements, to further detail topological constraints of function execution scheduling, we have the *controller*. This is an optional, block-level parameter that identifies which of the possible, available `controller`s in the current deployment we want to target to execute the scheduling policy of the current tag. Similarly to workers, we identify controllers with a label.

As mentioned above, a *controller* clause can have `topology_tolerance` as optional parameter. When deploying controllers and workers, users can label them with the topological *zone* they belong in[2]. Hence, when the designated `controller` is unavailable,

[1]The kind of computational resources that determine the `overload` option depends on the APIs provided by a given serverless platform. For example, in our prototype in Section 5.3.1 we consider a worker label `overload`ed when the related invokers are declared "unhealthy" by the OpenWhisk APIs, which use memory consumption and CPU load.

[2]Zone labels of controllers and workers are not used in tAPP scripts, which only specify co-location constraints, i.e., requests to consider workers in the same zone of a given controller. Zone labels are used by the infrastructure to implement the tAPP constraints.

tAPP can use this topological information to try to satisfy the scheduling request by forwarding it to some alternative controller.

The `topology_tolerance` parameter specifies what workers an alternative controller can use. Specifically, `all` is the default and most permissive option and imposes no restriction on the topology zone of workers; `same` constrains the function to run on workers in the same zone of the faulty controller (e.g., for data locality); `none` forbids the forward to other controllers. As an example, we could take advantage of the topology zones and rewrite the previous tAPP script from Listing 5.1 for the `couchdb_query` tag. e.g.,

```
- couchdb_query:
  - controller: DBZoneCtl
    workers:
     - set: local
       strategy: random
    topology_tolerance: same
  followup: default
```

this way it is guaranteed that the function will be executed always on the workers in the same zone of the database. Lastly, tAPP lets users express a selection strategy for policy blocks. This is represented by the optional *strategy* fragment of the *tag* rule. By default, when we omit to define a *strategy* policy for blocks, tAPP allocates functions following the blocks from top to bottom—i.e., `best_first` is the default policy. Here, for example, setting the `strategy` to `random` captures the simple load-balancing strategy of uniformly distributing requests among the available controllers.

5.3.1 tAPP in OpenWhisk

We modified OpenWhisk to support tAPP-based scheduling. In particular, to manage the deployment of components, we pair OpenWhisk with the popular and widely-supported container orchestrator Kubernetes.

The implementation entailed the creation and inclusion in the existing architecture of OpenWhisk of new components—e.g., a *watcher* service, which informs the gateway and the controllers on the current status of the nodes of the platform—and the extension

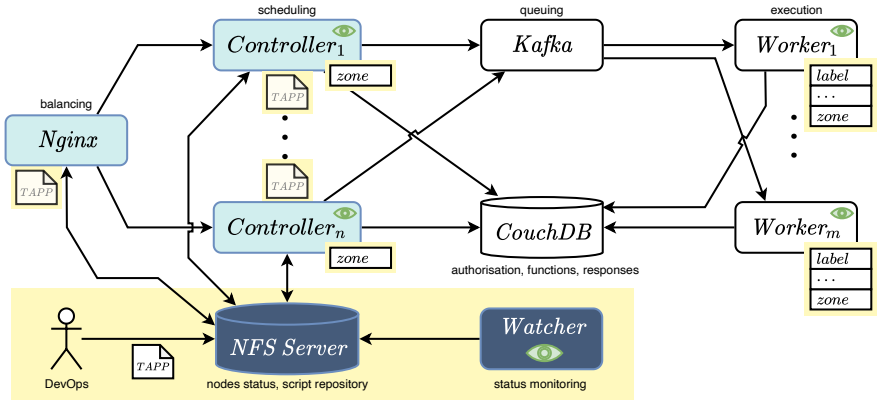


Figure 5.4: Architectural view of our OpenWhisk extension. We highlight in light blue the existing components of OpenWhisk we modified and in yellow the new ones we introduced.

of existing ones with new functionalities—e.g., to capture topological information at the level of workers and controllers, to enable live-reloading of tAPP policies, and let controllers and gateways follow tAPP policies depending on topological zones.

Figure 5.4 depicts the architecture of our OpenWhisk extension, where we reuse the *Workers* and the *Kafka* components, we modify *Nginx* and the *Controllers* (light blue in the picture), and we introduce two new services: the *Watcher* and the *NFS Server* (in the highlighted area of Figure 5.4).

The modifications mainly concert letting Nginx and Controllers retrieve and interpret both tAPP scripts and data on the status of nodes, to forward requests to the selected controllers and workers. Concerning the new services, the Watcher monitors the topology of the Kubernetes cluster and collects its current status into the NFS Server, which provides access to tAPP scripts and the collected data to the other components.

Topology-based Worker Distribution

To associate labels with pods, we exploit the topology labels provided by Kubernetes. These labels are names assigned to nodes and they are often used to orient pod allocation. Labels offer an intuitive way to describe the structure of the cluster, by annotating their zones and attributes. In Figure 5.4 we represent labels as boxes on the side of the controllers and workers.

Since OpenWhisk does not have a notion of topology, all controllers can schedule all functions on any available worker. Our extension unlocks a new design space that administrators can use to fine-tune how controllers access workers, based on their topology. At deployment, DevOps define the access policy used by all controllers. Our investigation led us to identify four topological-deployment access policies:

- the *default* policy is the original one of OpenWhisk, where controllers have access to a fraction of all workers' resources. This policy has two drawbacks. First, it tends to *overload* workers, since controllers race to access workers without knowing how the other controllers are using them. Second, it gives way to a form of *resource grabbing*, since controllers can access workers outside their zone, effectively taking resources away from "local" controllers;

- the *min_memory* policy is a refinement of the *default* policy and it mitigates overload and resource-grabbing by assigning only a minimal fraction of the worker' resources to "foreign" controllers. For example, in OpenWhisk the resources regard the available memory for one invocation (in OpenWhisk, 256MB). When workers have no controller in their topological zone, or no topological zone at all, we follow the default policy. Also this policy has a drawback: it can lead to scenarios where smaller zones quickly become saturated and unable to handle requests;

- the *isolated* policy lets controllers access only co-located workers. This reduces overloading and resource grabbing but accentuates small-zone saturation effects;

- the *shared* policy allows controllers to access primarily local workers and let them access foreign ones after having exhausted the local ones. This policy mediates between partitioning resources and the efficient usage of the available ones, although it suffers a stronger effect of resource-grabbing from remote controllers.

In case no tAPP script is available, controllers resort to their original, hard-coded logic (explained in Section 3.2.2) but still prioritise scheduling functions on co-located workers.

5.3.2 Deploying tAPP-based OpenWhisk

The standard way to deploy OpenWhisk is by using the Docker images available for each component of the architecture—this lets developers choose the configuration that suits their deployment scenario, spanning single-machine deployments, where all the components run on the same node, and clustered (e.g., via Kubernetes) deployments, e.g., assigning a different node to each component. Since we modified the Controller component of the architecture (see Section 5.3.1), we built a new, dedicated Docker image and published it on DockerHub[3], so that it is generally available to be used in place of the vanilla controller. Both for reproducibility and reliability, we automate all the levels of the deployment steps: the provisioning of the virtual machines (VMs) and both the deployment of Kubernetes and of (our extended version of) OpenWhisk. We programmatically provision VMs using the Google Cloud Platform via a Terraform[4] script. Since this script is tied to a specific topology, we provide more information on it when describing our experiments in Section 5.4.1.

We wrote Ansible[5] scripts instead to automatically deploy the Kubernetes cluster. Given the VMs where one wants to deploy Kubernetes on and their designated roles (workers, etc.), our Ansible scripts configure each VM by installing the dependencies required for Kubernetes, deploy the control-plane on the designated master VM with the `kubeadm` tool, and make the other VMs join the cluster as worker nodes (again with the `kubeadm` tool).

Once the Kubernetes cluster is up and running, we use the Helm[6] package from `openwhisk-deploy-kube` [83], that we forked to implement a tAPP-specific package for the installation with our custom controller image. This automatically deploys every component on a Kubernetes cluster and allows the user to parameterize the configuration of the deployment; specifically, we configure the deployment to select our tAPP-based controller image.

All Terraform and Ansible scripts are publicly available at `https://github.com/giusdp/ow-gcp`.

[3] `https://hub.docker.com/r/mattrent/ow-controller`.
[4] `https://www.terraform.io/`.
[5] `https://www.ansible.com/`.
[6] `https://helm.sh/`.

5.4 Case Study

As a final illustration of the tAPP language, we show and comment on the salient parts of a tAPP script—reported in Figure 5.5—that captures the scheduling semantics of the case in Figure 5.2.

```
1  - critical:
2    - controller: LocalCtl_1
3      workers:
4        - set: edge
5          strategy: random
6    followup: fail
7  - machine_learning:
8    - controller: CloudCtl
9      workers:
10       - set: cloud
11     topology_tolerance: same
12   followup: default
13 - default:
14   - controller: LocalCtl_1
15     workers:
16       - set: internal
17         strategy: random
18       - set: cloud
19         strategy: random
20     strategy: best_first
21   - controller: LocalCtl_2
22     workers: # same as above
23     strategy: best_first
24   strategy: random
```

Figure 5.5: A tAPP script that implements the scheduling semantics of the case study in Section 5.1 (Figure 5.2).

In the script, at lines 1–6, we define the tag associated to critical (ⓘ) functions: only *LocalCtl_1* can manage their scheduling, they can only execute on #edge workers (W_1,...,W_i in Figure 5.2), and no other policy can manage them (`followup: fail`). At line 5 we specify to evenly distribute the load among all edge workers with `strategy: random`.

At lines 7–12, we find the tag of the machine_learning ( ) functions. We define *CloudCtl* as the controller and consider all #cloud workers (W_{k+1},...,W_j in Figure 5.2) as executors. Notice that at line 12 we specify to use the `default` policy as the `followup`, in case of failure. The interaction between the `followup` and the `topology_tolerance` (line 11) parameters makes for an interesting case. Since the `topology_tolerance` is (the) `same` (zone of the controller *CloudCtl*), we allow other controllers to manage the scheduling of the function (in the `default` tag) but we continue to restrict the execution of machine-learning functions only to workers within the `same` zone of *CloudCtl*, which, here, coincide with #cloud-tagged workers.

Lines 13–24 define the special,`default` policy tag, which is the one used with tag-less functions (here, our generic ones) and with failing tags targeting it as their `followup` (as seen above, line 12). In particular, the instruction at line 24 indicates that the `default` policy shall randomly distribute the load on both worker blocks (lines 14–20 and 21–23), respectively controlled by *LocalCtl_1* and *LocalCtl_2*. Since the two blocks at lines 14–20 and 21–23 are the same, besides the `controller` parameter, we focus on the first one. There, we indicate two sets of valid workers: the #internal ones (line 16, W_{i+1},...,W_k in Figure 5.2) and the #cloud ones (as seen above, for lines 9–10). The instruction at line 20 (`strategy: best_first`) indicates a precedence: first we try to run functions on the #local cluster and, in case we fail to find valid workers, we offload on the #cloud workers—in both cases, we distribute the load randomly (lines 17 and 19).

5.4.1 Case Study Implementation

We now evaluate our contribution by presenting a cloud-edge-continuum case study, taken from the literature, to both demonstrate how one can use tAPP to meet topology-aware functional requirements and how existing serverless solutions—where no topological information is used by the function scheduler, like vanilla OpenWhisk—fail in complying with those requirements.

The case study we consider is a simplification of the architecture described in Section 5.1, which we depict in Figure 5.6. It is a serverless cloud-edge deployment of the system described in [48, 49], consisting of a power transformers' anomaly detection application. Each power transformer to be monitored is equipped with six accelerometers

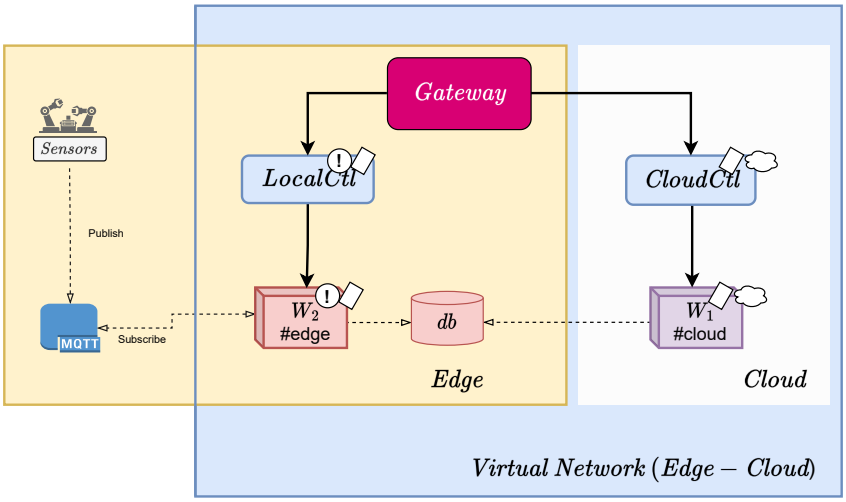


Figure 5.6: Use case architecture: the services were separated in two zones, named *Edge* and *Cloud*, and connected using two different networks, a local network corresponding to the *Edge* zone and a virtual private network used for the OpenWhisk cloud-edge deployment.

that produce data at a frequency of 10kHz. At every minute, the data produced by the six sensors are collected for one second. For each set of data produced by one sensor, two features are extracted: FCA (based on Frequency Complexity Analysis) and DET (based on Vibration Stationarity Analysis). These features are then combined in two vectors (one with the six FCA features and one with the six DET features) and classified following machine learning techniques.

In the case study, we assume that the sensors communicate their data through an IoT-specific protocol to a message broker accessible only from machines within the same local network, named *Edge*. This prevents public access to the broker to protect it, e.g., from denial of service attacks. While data gathering happens locally, the elaboration of the data requires powerful resources. For this reason, when local resources are available, we run these analyses locally, otherwise, we run them in the Cloud.

In our serverless deployment of the use case, we have assumed that the sensors communicate their data via a standard IoT protocol, namely MQTT [75], and that the workflow is implemented as a pipeline of three separate functions:

- **data-collection**, which contacts an MQTT broker and subscribes to six topics

```
- default:
  - workers:
      - set:
- MQTT:
  - controller: LocalCtl
    workers:
      - set: edge
    topology_tolerance: none
    followup: fail
```

```
- DB:
  - workers:
    - wrk: W_2
      invalidate: capacity_used 50%
    - wrk: W_1
    strategy: best-first
- Cloud:
  - controller: CloudCtl
    workers:
      - set: cloud
    topology_tolerance: none
    followup: fail
```

Figure 5.7: Script used in the tAPP-based use case deployments.

(one for each sensor), receives the corresponding data and stores it in a local database;

- **feature-extraction**, which queries the database for the collected data and extracts relevant features;
- **feature-analysis**, which receives the extracted features and performs the classification task.

To perform the evaluation, we deploy the platform and services in two different zones, as represented in Figure 5.6: a *cloud* zone containing the Kubernetes master node, one OpenWhisk controller and one worker, and an *edge* zone containing the MQTT broker, the database, one OpenWhisk controller and another worker. Functions

can access the MQTT broker only from the edge zone, while they can access the database from the entire cluster, and accordingly, from both workers.

We use a total of 6 separate VMs. Specifically, we use 3 e2-medium instances (2 vCPUs, 4GB of RAM) on Google Cloud Platform, located in the Belgian data centre (europe-west1-b) for the *cloud* zone. One of these VMs acts as the Kubernetes master node and the other 2 as the cloud Controller and the cloud worker. The other 3 VMs for the *edge* zone are hosted on Digital Ocean in the Frankfurt data centre. Specifically, one edge VM hosts the edge Controller, one the edge worker, and one both the database and the MQTT broker. All VMs sport 2 vCPUs and 4GB of RAM each. We connect the VMs using two separate networks:

- a virtual private network, containing the entire Kubernetes cluster (i.e., Kubernetes master node, cloud and edge Controllers, cloud and edge workers) and the database;
- a local network, containing the entire edge zone (i.e., edge worker, edge Controller, MQTT broker and database).

We simulate the sensors via a Python script running on the edge VM hosting the MQTT broker. To mimic the real system that inspired our experiments, we kept the workflow frequency at one invocation per minute and the push frequency of the sensors at 10kHz (i.e., ten thousand tuples pushed per second, per sensor). Since the influence on the experiment of the specific machine learning model used for the analysis is immaterial, we implement these steps as functions that receive/gather the data from the database, perform a heavy workload (matrix multiplications), and then generate a predetermined response.

Our first experiment is about the deployment of the case study by using vanilla OpenWhisk. Repeating the experiment 10 times we observed that randomly the system would be deployed in a way that the invocation would either work as intended or fail every invocation of the **data-collection** function, invalidating the entire pipeline. The reason for this behaviour is that OpenWhisk randomly assigns identifiers to workers and use them to make scheduling decisions (cf. the platform strategy, Section 5.3). If the algorithm marks the worker that cannot reach the MQTT broker as the one where to send **data-collection**, this function will consistently fail to connect to the MQTT broker.

As shown in Figure 5.7, we overcome this limitation with tAPP by specifying the appropriate function allocations:

- we tag **data-collection** (ⓘ) with *MQTT*, so that the *LocalCtl* controller decides their scheduling, which means it allocates the functions only in the *edge* zone and on workers located in the same network as the MQTT broker. By setting the topology tolerance to *none* we forbid forwarding to the other controllers and ensure this function only runs on the *edge* worker;
- we tag **feature-extraction** (◇) with *DB*. In this case, we do not define a controller, but we specify a list of workers so that the controller always picks the worker in the *edge* zone first. For this function, we specify a priority among the workers since they can all reach the database, preferring to use the *edge* worker—until it exceeds a capacity of 50%—since it is the closest to the *db*.
- we tag **feature-analysis** (☁) with *Cloud*. Similarly to the data-collection function, we specify a specific controller, *CloudCtl*, and a tolerance of *none* to use the *cloud* zone exclusively. In this way, we simulate a situation where machine learning tasks would be moved away from the edge machines, which usually have fewer resources or stricter requirements.

Experimental Data We recorded the performance of the system using the vanilla and tAPP-based OpenWhisk variants. The starting version for both tAPP-based OpenWhisk and the vanilla OpenWhisk in the experiments originate from commit `aa7e6e2` of the official Apache OpenWhisk repository.[7] For the vanilla version, we add a logging mechanism to record the scheduling times for the invocations.

We repeated the deployments of both versions of the platform 10 times. Once deployed, we tested the use case by performing 100 sequential invocations of the pipeline with an interval of 1 minute between each invocation as described in the use case, and recording the latencies of the function invocations, i.e., the time passed between a request to a function and a response (successful or failing), and the scheduling time, i.e., the time the scheduler takes to pick a worker for the function invocation. We performed these test runs with Locust[8], a load testing tool.

[7] `github.com/apache/openwhisk/commit/aa7e6e2af196ac017ae4b9ea36656bec868a9931`
[8] `https://locust.io/`

We observed that vanilla OpenWhisk picks the cloud worker as the main worker for the **data-collection** function a total of 8 out of 10 times.[9] In these cases, the pipeline fails to connect to the MQTT broker and the **data-collection** stalls until the the timeout mechanism is triggered after 60 seconds. The other 2 functions are not invoked as the function pipeline stops at its first step. The tAPP-based deployment has instead a success rate of the pipeline invocation of 100%, with the **data-collection** function being always allocated to the edge worker, resulting in consistent performance.

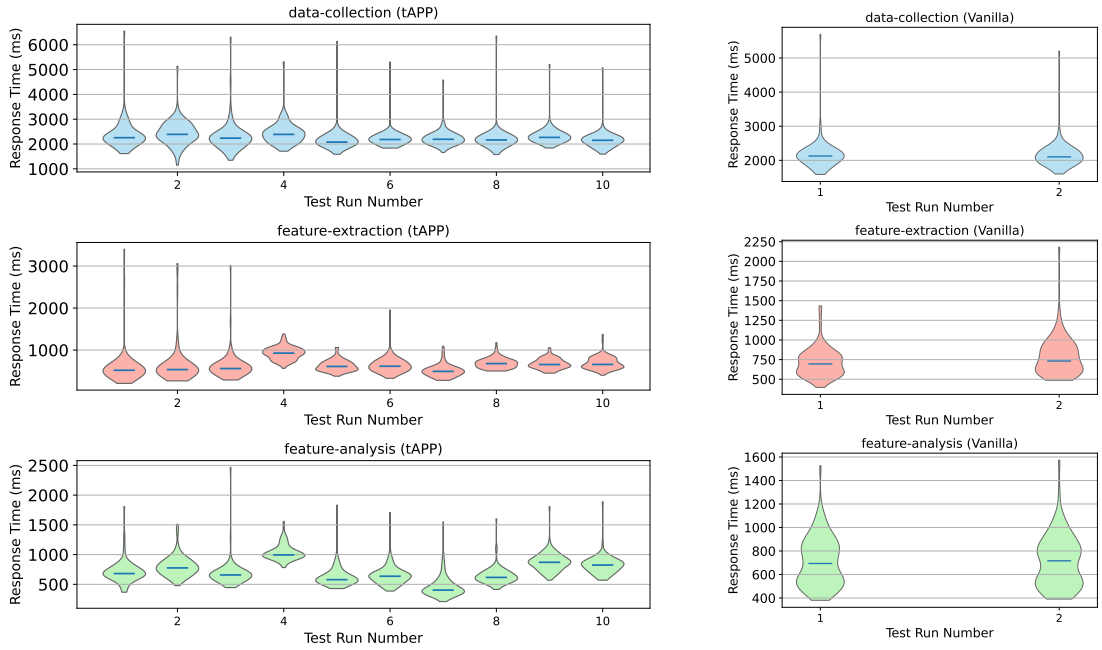


Figure 5.8: Latencies in tAPP-based OpenWhisk (left) and vanilla OpenWhisk (right).

Figure 5.8 shows the violin plots related to the response time of the 3 functions in all the 10 tAPP-based OpenWhisk deployments and the 2 functioning vanilla OpenWhisk deployments. The response times cluster around 2000 ms for **data-collection**, less than 1000 ms for **feature-extraction**, and between 500 and 1000 ms for **feature-analysis** for both the tAPP and vanilla OpenWhisk. In both scenarios, there are outliers,

[9]We remark that, while we were expecting a failure in 50% of the cases, the chosen worker from the vanilla scheduler was almost always the cloud worker. It is beyond the scope of this thesis to ascertain if this was due to a statistical anomaly or to OpenWhisk's internal decisions (e.g., decisions based on a hash from the namespace and the function name) that favour particular deployment choices.

representing the high response times caused by cold starts when the functions are first invoked.

For completeness, we report in Table 5.1 the aggregated mean, median, tail latency (i.e., 99 latency percentile), and standard deviation of the tAPP-based and the 2 functioning vanilla OpenWhisk deployments—for compactness, we use the icons ⓘ, ◇, ☁ to represent resp. the data-collection, feature-extraction, and feature-analysis functions. As can be seen, in case the deployment of vanilla OpenWhisk did allow the scheduling of the data-collection function, the response times were similar.

	tAPP-based				**Vanilla**			
Function	Mean	Median	Tail	Std. Dev.	Mean	Median	Tail	Std. Dev.
ⓘ	2272	2231	2543	345	2148	2114	2522	361
◇	652	629	981	91	749	727	1097	141
☁	729	716	830	113	707	691	975	111

Table 5.1: Latency (ms) of the tAPP-based and vanilla OpenWhisk deployments.

Finally, Table 5.2 reports the mean, median, tail, and standard deviation of the scheduling time of the tAPP-based and vanilla deployments, i.e., the time it takes the controller to schedule a function from when the function invocation is received to when the scheduling logic is executed and a worker is chosen. From the results, we can conclude that the performance of vanilla OpenWhisk and tAPP are comparable—the average scheduling time is below 2 ms for both the tAPP and vanilla OpenWhisk deployments.

Function	**tAPP-based**				**Vanilla**			
	Mean	Median	Tail	Std. Dev.	Mean	Median	Tail	Std. Dev.
ⓘ	1.42	1.29	2.60	0.57	1.78	1.54	3.75	1.22
◇	1.62	1.43	4.23	0.93	2.0	1.74	3.82	1.20
☁	1.68	1.68	2.85	0.64	1.95	1.73	3.82	1.02

Table 5.2: Scheduling time (ms) of the tAPP-based and vanilla OpenWhisk deployments.

5.4.2 Overhead Analysis

We investigate the performance of our tAPP extension looking at the overhead it exerts w.r.t. vanilla OpenWhisk. To this aim, we run a set of experiments using a benchmark suite for serverless platforms, called *ServerlessBench* [128]. Running the tests, we measure both the scheduling time (the time taken to pick on which worker to allocate a function) and the request-reply latency of functions.

ServerlessBench consists of 12 test cases exploring several metrics of serverless computing, like communication efficiency and startup latency (cold starts). Of these 12 test cases, 6 apply to OpenWhisk: 3–6, 9, and 10. Of these, case 4 consists of 4 subtests, each with different example applications requiring different resources (i.e., databases, Alexa devices). Due to these special requirements, we exclude case 4 from our analysis and focus on the remaining 5.

We run cases on both vanilla OpenWhisk and tAPP-based OpenWhisk deployments. For the tAPP variant, we use a simple tAPP script with default settings that make the tAPP variant behave like vanilla OpenWhisk. In this way, we perform a same-settings comparison of the two versions of the platform. We run each case 5 times to obtain consistent data.

Case 3 This case focuses on function composition, obtained by invoking a long pipeline of functions. The functions in the pipeline are instances of the same one, written in JavaScript, which increments by one the input value and returns it. We create a "sequence" (the OpenWhisk built-in pipeline construct) with 50 of these functions so that the platform invokes them in sequence, passing the output of one function as the input of the next one. We invoke the pipeline 20 times to have a total of 1000 invocations in a single test run. We report the latencies and scheduling times in Figure 5.9. From the results, the time required to pick a worker is essentially the same for both versions of OpenWhisk. Similarly, the total latency of the invocations—each spanning the entire sequence and corresponding to the time required to execute all the functions in the pipeline—is comparable, with the majority falling between the 17 to 24 seconds range. In particular, vanilla OpenWhisk shows a higher fluctuation in the total latencies with a higher standard deviation of 2435.11 ms (and mean of 20096.86 ms), while tAPP had a more stable performance (standard deviation of 1139.1 ms and

mean of 18901.26 ms).

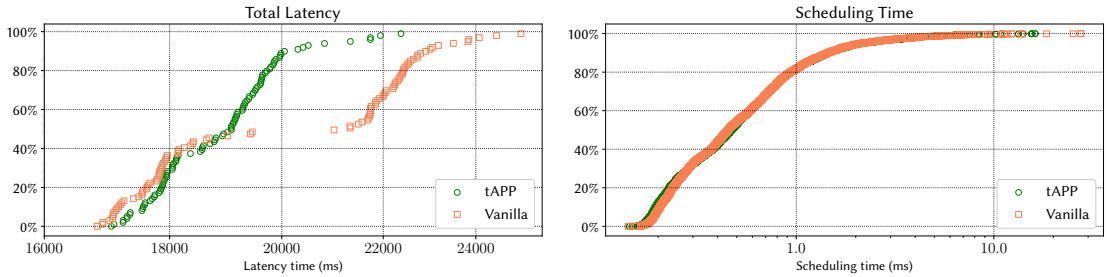


Figure 5.9: Test Case 3 latencies (left) and scheduling time (right).

Case 5 This case focuses on data transfer costs, using a sequence of 2 functions where a file of 32KB is passed to the first function, which passes it to the second one. We invoke the sequence 500 times to have a total of 1000 invocations in a single test run, as in the previous case. We show the latencies and scheduling times in Figure 5.10. Consistently with the observations of case 3, we have negligible differences in scheduling times between the two platforms, and the total latencies are stable between 500 ms and 2000 ms for both platforms, with some outliers reaching 4000 ms due to cold starts. Regarding invocation latency, vanilla OpenWhisk has a slightly higher standard deviation of 357.27 ms and a mean of 967.55 ms, while tAPP has a standard deviation of 186.67 ms and a mean of 855.82 ms.

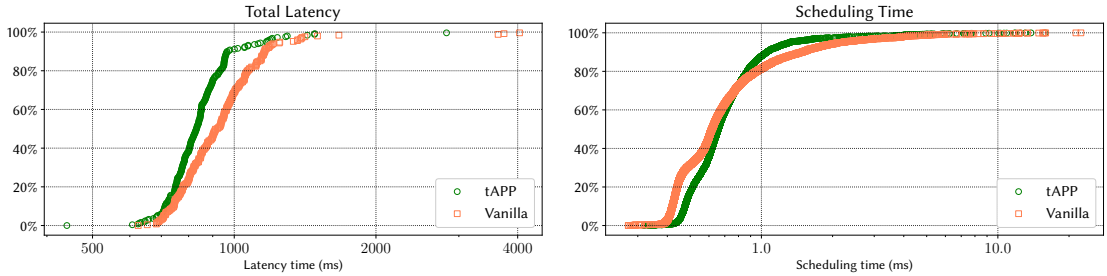


Figure 5.10: Test Case 5 latencies (left) and scheduling time (right).

Both cases 6 and 9 focus on startup latency, tested by invoking functions while inducing cold starts.

Case 6 In case 6, we invoke a Java function that uses a custom Java container runtime for OpenWhisk, which incurs long initialisation times. To enforce cold starts, one must either invoke the function after 10 minutes since the last invocation—OpenWhisk keeps a function's container alive for 10 minutes since the last call to reduce cold starts—or manually stop the function container on the worker. To avoid interfering with the platform's internal dynamics, we preferred the first option and, to keep the test times reasonable, we opted to perform 10 invocations for each test run. We present the latencies and scheduling times in Figure 5.11. The results are in line with the previous test cases, with negligible differences in scheduling times between the two platforms. The total latencies are stable between 2000 ms and 4000 ms with a mean of 2943.87 ms for tAPP OpenWhisk and a standard deviation of 476.39 ms, and a mean of 3494.07 ms and a standard deviation of 672.62 ms for vanilla OpenWhisk.

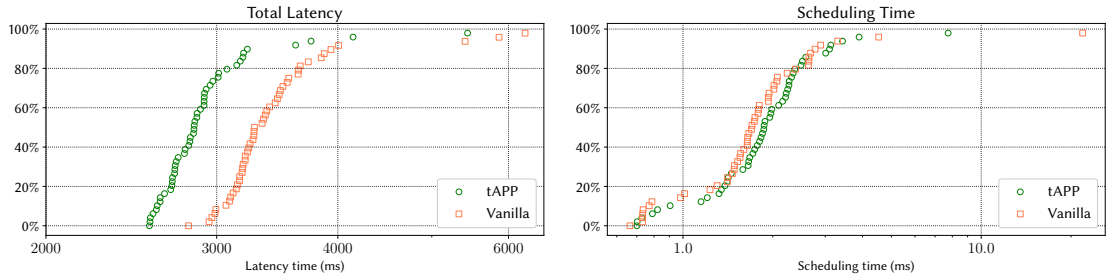


Figure 5.11: Test Case 6 latencies (left) and scheduling time (right).

Case 9 This test case covers a concurrent startup scenario to analyse how auto-scaling impacts function startup. We use the Java and C functions of the case, and we run 10 invocations of the functions in a single test run. For each function, we send 40 requests simultaneously, once with a maximum concurrency limit (i.e., the maximum number of concurrent invocations of the same function per container) set to 1 and once set to 40, effectively resulting in four subtests. We show the latencies and scheduling times in Figure 5.12 and Figure 5.13. In all subtests, the two platforms have similar performance. We report in Table 5.3 the aggregated mean, median, tail latency (i.e., 95 latency percentile), and the standard deviation of the tAPP-based and vanilla OpenWhisk deployments.

	Conc.	tAPP-based				Vanilla			
		Mean	Median	Tail	S. Dev.	Mean	Median	Tail	S. Dev.
C	1	8892	8692	10195	693	8808	8524	10421	776
Java	1	9861	9602	11739	931	10202	10245	11665	1026
C	40	10098	10187	11731	915	10171	9962	12563	1289
Java	40	8925	8767	10380	742	9433	9043	11361	1078

Table 5.3: Statistics of the tAPP-based and vanilla OpenWhisk deployments forf Test Case 9 (ms).

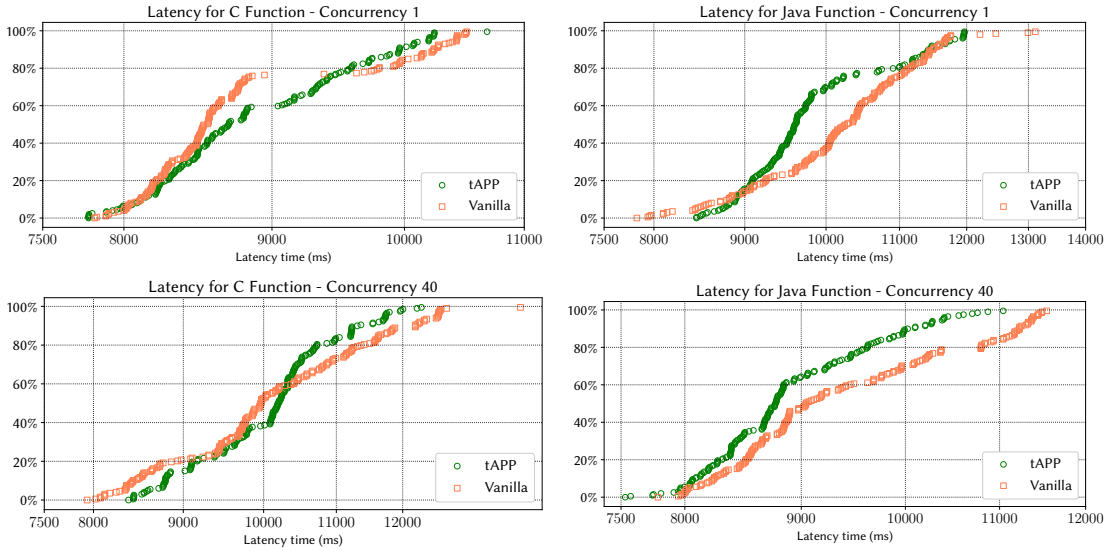


Figure 5.12: Test Case 9 latencies for the four subtests: C and Java functions with concurrency 1 (top-left and top-right), C and Java functions with concurrency 40 (bottom-left and bottom-right).

Case 10 This test case focuses on the effect of implicit state with a Java function that performs image resizing using a custom Java runtime. The function takes advantage of "warm" containers by re-using the implicit state of the Java runtime. We invoke the function 1000 times in a single test run, reporting in Figure 5.14 the latencies and scheduling times. Similarly to the previous test cases, the scheduling time differences between the two platforms are negligible. The request-reply latencies go from below 500 ms to more than 4000 ms for both platforms, with a mean of 610.67 ms and a standard deviation of 256.67 ms for tAPP OpenWhisk and a mean of 811.44 ms and a standard deviation of 431.69 ms for vanilla OpenWhisk.

Overall, these experiments demonstrate that the scheduling time for our tAPP

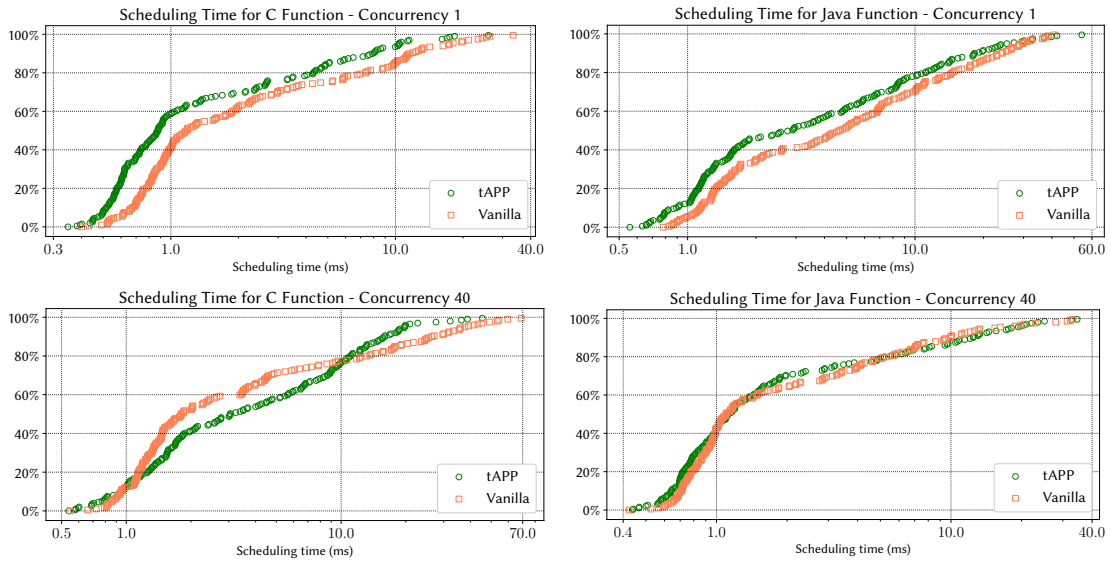


Figure 5.13: Test Case 9 scheduling time for the four subtests: C and Java functions with concurrency 1 (top-left and top-right), C and Java functions with concurrency 40 (bottom-left and bottom-right).

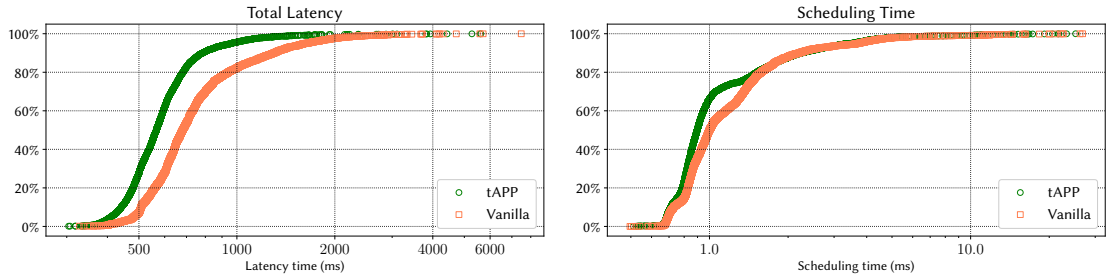


Figure 5.14: Test Case 10 latencies (left) and scheduling time (right).

prototype is comparable to, if not better than, the vanilla implementation. Consequently, our solution does not present any significant performance drawbacks when compared to vanilla OpenWhisk.

5.5 Conclusion

We introduced tAPP, a declarative language that provides users with finer control over the scheduling of serverless functions. Being topology-aware, tAPP scripts can

restrict the execution of functions within zones and help improve the performance (e.g., exploiting data or code locality properties), security, and resilience of serverless applications. To validate our approach, we presented a prototype tAPP-based serverless platform, developed on top of OpenWhisk, and we used it to show that tAPP allows for an easy deployment of cloud-edge serverless systems with typical topology-aware scheduling constraints that cannot be guaranteed by standard vanilla OpenWhisk deployments. As future work we plan to expand our range of tests both to include other aspects of locality (e.g., sessions) and specific components of the platform (e.g., message queues, controllers). We also intend to formalise the semantics of tAPP, e.g., building on existing "serverless calculi" [39, 51]. This is a stepping stone to mathematically reason on scheduling policies and formally prove they provide desirable guarantees.

Chapter 6

Affinity-aware Serverless Scheduling

6.1 Introduction

The breadth of the design space of serverless scheduling policies is witnessed by the growing literature focused on techniques that mix one or more of these locality principles to increase the performance of function execution, assuming some locality-bound traits of functions [22, 56, 15, 52, 60, 101]. Besides performance, functions can have functional requirements that the scheduler could consider. For example, users might want to ward off allocating their functions alongside "untrusted" ones—common threat vectors in serverless are limited function isolation and the ability of functions to (surreptitiously) gather weaponisable information on the runtime, the infrastructure, and the other tenants [13, 115, 6, 29]. Although one can mix different principles to expand the profile coverage of a given platform-wide scheduler policy, the latter hardly suits all kinds of scenarios. This shortcoming was one of the motivation for our domain-specific, platform-agnostic, declarative language APP (and later the extension tAPP). Thanks to APP, the same platform can support different scheduling policies, each tailored to meet the specific needs of a set of related functions. As mentioned in Chapter 1, we study the addition of affinity and anti-affinity constraints at the FaaS level by proposing a new *affinity-aware* extension, called aAPP, after observing that other cloud platforms like IaaS and CaaS support affinity and anti-affinity constraints for workload allocation, which FaaS platforms lack native mechanisms for.

Example

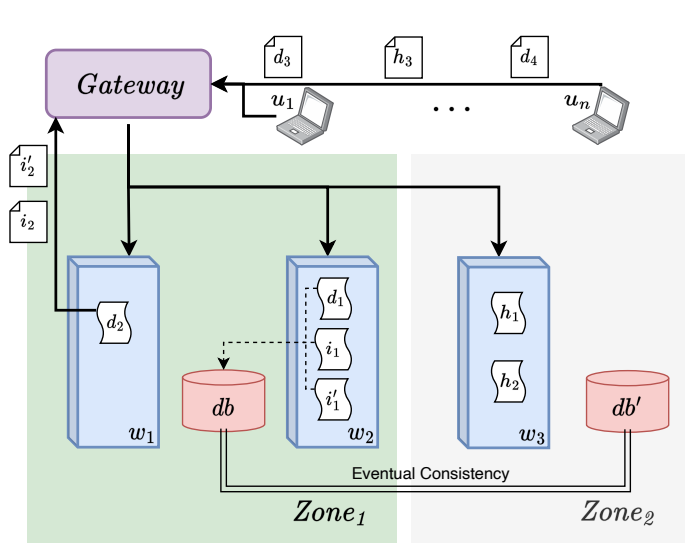
We introduce and motivate aAPP-based affinity-aware FaaS scheduling policies with an example. We have a *divide-et-impera* data-crunching serverless application implemented through two companion functions. The first, invoked by the users, is called *divide*. Its task is to split some data into chunks, store them in a database, and invoke instances of the second function. The second function, which the *divide* invokes for each stored chunk, is called *impera*. Its task is to retrieve a chunk of data from the database and process it.

We run the above functions on the FaaS infrastructure depicted on the left of Figure 6.1. The infrastructure includes two zones (e.g., separate regions of a cloud provider) and it has a *Gateway* that decides on which worker to allocate the execution of the functions. The infrastructures also incldues three workers: w_1 and w_2 in $Zone_1$ and w_3 in $Zone_2$. Each zone hosts an instance of an eventually-consistent distributed database [114], used by the functions running in that zone—eventually-consistent systems are the preferred choice for (FaaS) scenarios like our example, where one favours throughput and availability w.r.t. e.g., overall data consistency [11].

In Figure 6.1, we represent function allocation requests with labelled document icons sent towards the *Gateway*. Note that the users (the laptop icons in Figure 6.1) launch the *divide* function (e.g., d_3) and while the running *divide* invokes the *impera* functions (e.g., d_2 requesting i_2 and i'_2).

Our FaaS infrastructure executes additional applications besides the one above. In Figure 6.1, we represent these requests with the labels h_1, h_2, and h_3 which are compute-intensive functions—called *heavy*—that use a high amount of computational resources of the worker running them.

Given this context, a first example of an affinity-aware scheduling policy is to avoid the co-occurrence of the *divide* and *impera* functions with the *heavy* ones. In this way, we can improve the performance of *divide* and *impera* by avoiding resource contention with the *heavy* functions. Another improvement regards the interaction with the database. The eventual-consistency behaviour of the database entails possible delays to synchronise the instances. Waiting for synchronisation is necessary only when the functions accessing the database connect to different database instances. Moreover, to further reduce delay, we can exploit the principle of *session locality* and




```
1  - d:
2    - workers: *
3      affinity:
4        - !h
5  - i:
6    - workers: *
7      affinity:
8        - !h
9        - d
10 - h:
11   - workers: *
12     affinity:
13       - !d
14       - !i
```

Figure 6.1: Example of a FaaS infrastructure (left) and an aAPP script (right).

let functions running on the same worker share the same connection with the database. This affinity-aware scheduling policy places *impera* functions only on workers that already host *divide* functions and avoid the overhead of re-establishing new connections.

These constraints can be encoded in aAPP as shown in the script in Figure 6.1. This code has three top-level items: `d`, `i`, and `h`. These are tags that identify policies, each describing the scheduling logic of a set of related functions. In the example, the tag `d` describes the logic for the *divide* functions while `i` and `h` target respectively the *impera* and *heavy* ones. The line `workers: *` found under all tags indicates that their related functions can use any of the available workers. From the top, under tag `d`, we use the `affinity` clause, introduced by aAPP, to specify that `d`-tagged functions should *not* be scheduled on a worker that currently hosts *heavy* functions (`!h`). Specifically, this is an example of *anti-affinity*, where we prevent the allocation of the tagged functions (e.g., `d`) on a worker that already hosts any anti-affine function (e.g., tagged `h`). Tag `i` declares the same anti-affinity for *heavy* functions, but it also indicates that `i`-tagged functions are *affine* with `d`-tagged ones. Affinity means that we can schedule a function on a candidate worker only if it currently hosts the former's affine functions. In the example, we use affinity to have *impera* functions run in the same worker of *divide* functions. Finally, we use tag `h` to complement the anti-affinity relation expressed in the previous tags, i.e., the *heavy* functions are anti-affine with both `d` and `i` functions

and shall not be scheduled in workers that already host any of the latter.

Structure of the chapter. In Section 6.2 we present aAPP, an APP extension with (anti)-affinity constraints. In Section 6.3, we discuss our extension of the Apache OpenWhisk scheduler (i.e., a popular open-source FaaS solution) to support aAPP. We use our prototype to show, in Section 6.4, that using aAPP in affinity-bound scenarios (like the one presented above) leads to an appreciable reduction in latency. Moreover, by using microbenchmarks, in Section 6.5, we show that the overhead of supporting aAPP-based affinity is negligible. We draw our conclusions in Section 6.6.

6.2 The aAPP Language

In this section, we present aAPP, our extension of the FaaS function scheduling language APP [31, 30] with affinity and anti-affinity constraints.

We report in Figure 6.2 the syntax of aAPP. From here on, we indicate syntactic units in *italics*, optional fragments in *grey*, terminals in `monospace`, and lists with $\overline{bars}$[1]. The idea behind aAPP is that functions have associated a tag that identifies some scheduling policies. An aAPP script represents: *i)* named scheduling policies identified by a *tag* and *ii)* policy *block*s that indicate either some collection of workers, each identified by a worker *id*, or the universal `*`. To schedule a function, we use its tag to retrieve the scheduling policy that includes one or more blocks of possible workers. To select the worker, we iterate top-to-bottom on the blocks. We stop at the first block that has a non-empty list of valid workers and then select one of those workers according to the strategy defined by the block (described later).

Each tag can define a `followup` clause, which specifies what to do if the policy of the tag did not lead to the scheduling of the function; either `fail`, to terminate the scheduling, or `default` to apply the special `default`-tagged policy. Each block can define a `strategy` for worker selection (`any` selects non-deterministically one of the available workers in the list; `best_first` selects the first available worker in the list), a list of constraints that `invalidate`s a worker for the allocation (`capacity_used` invalidates a worker if its resource occupation reaches the set threshold; `max_concurrent_invocations`

[1]While aAPP scripts are YAML-compliant, for presentation, we slightly stylise the syntax to increase readability. For instance, we omit quotes around strings, e.g., `*` instead of `"*"`.

invalidates a worker if it hosts more than the specified number of functions), and an `affinity` clause that carries a list containing affine tag identifiers *id* and anti-affine tags, represented by negated tag identifiers *!id*. aAPP is a minimal extension of APP adding the possibility to use this latter `affinity` construct that is not available in the original APP proposal.

As an example, Listing 6.1 shows an aAPP policy for functions tagged `f_tag`. The policy has two blocks. The former restricts the allocation of the function on the workers labelled `local_w1` and `local_w2` and the latter on `public_w1`. The first block specifies as invalid (i.e., which cannot host the function under scheduling) the workers that reach a memory consumption above 80%. Since the strategy is `best_first`, we allocate the function on the first valid worker; if none are valid, we proceed with the next block. The function has affinity with `g_tag` and anti-affinity with `h_tag`. Hence, a valid worker requires the presence of at least a function with tag `g_tag` and no functions with tag `h_tag`. If both the first and second blocks do not find a valid worker, the scheduling of the function `fail`s (instead of continuing with the `default` tag).

$$
\begin{array}{lcl}
 & & id \in Identifiers \qquad n \in \mathbb{N} \\
app & ::= & \overline{-tag} \\
tag & ::= & id : \overline{- \ block} \quad \texttt{followup} : f\_opt \\
block & ::= & \texttt{workers} : w\_opt \quad \texttt{strategy} : s\_opt \\
 & & \quad \texttt{invalidate} : \overline{- \ i\_opt} \quad \texttt{affinity} : \overline{- \ a\_opt} \\
w\_opt & ::= & * \mid \overline{- \ id} \\
s\_opt & ::= & \texttt{any} \mid \texttt{best\_first} \\
i\_opt & ::= & \texttt{capacity\_used} \ n\% \mid \texttt{max\_concurrent\_invocations} \ n \\
a\_opt & ::= & id \mid !id \\
f\_opt & ::= & \texttt{default} \mid \texttt{fail}
\end{array}
$$

Figure 6.2: aAPP syntax.

6.3 aAPP-based Apache OpenWhisk

We have implemented and validated an aAPP-based FaaS platform, obtained by extending the APP prototype of Apache OpenWhisk. The main intervention we performed to make the existing APP-based OpenWhisk architecture aAPP-compliant

```
- f_tag:
- workers:
    - local_w1
    - local_w2
    strategy: best_first
    invalidate:
    - capacity_used 80%
    affinity: g_tag,!h_tag
- workers:
    - public_w1
followup: fail
```

Listing 6.1: Example aAPP script.

consists of an extension of the *Controller* component. The extension adds a parser for the aAPP scripts and a new scheduler that handles the given policies, but the major challenge of implementing aAPP has been changing the Apache OpenWhisk's load balancer—the part of the Controller responsible for scheduling the functions—so that it keeps track of the functions allocated to all the workers. We introduced two lookup tables to implement this tracking functionality: the *activeFunctions* and the *activeTagActivations*. The first table associates the allocated functions (and their tags) to their host worker and allows the load balancer to verify affinity and anti-affinity constraints. The second table is an auxiliary one. Indeed, to update the *activeFunctions* table, we need to keep track of the state of the different function instances (possibly of the same function definition, so we cannot use their identifiers) by pairing their *activation id*s with their function identifiers; when we observe the termination of an active function, we look its function identifier up and remove that instance from the *activeFunctions* table—we detect instance terminations thanks to the messages workers send to notify the load balancer of their completion.

The scheduling algorithm following an aAPP script is straightforward. We present it in (Python-like) pseudo-code in Listing 6.2 and Listing 6.3. In Listing 6.2, the `schedule` function requires the name of the function to be scheduled (`f`), the map representing the infrastructure configuration (`conf`), the aAPP script encoded as a Python dictionary of objects (`aapp`), and a registry mapping the memory occupation and the tag for every function (`reg`). The configuration of a worker is assumed to be a map, denoting with

```
1  def schedule(f, conf, aapp, reg):
2    (memory, tag) = reg[f]
3    blocks = aapp[tag].blocks # get the blocks
4    if aapp[tag].followup != 'fail':
5      blocks += aapp['default'].blocks # add default tag blocks
6    for block in blocks:
7      if '*' in block['workers']:
8        block['workers'] = conf.keys
9      workers = [ for worker in block['workers'] if valid(f,worker,conf,reg,block)]
10     if len(workers) > 0: # if at least one valid worker is found
11       if block['strategy'] == 'best_first':
12         return workers[0]
13       elif block['strategy'] == 'any':
14         return random.choice(workers)
15   raise Exception('Function not schedulable')
```

Listing 6.2: The pseudo-code of the `schedule` function.

`fs`, `memory_used`, and `max_memory` respectively the list of functions already scheduled on the node, the memory allocated for those functions, and the total amount of memory of the worker. Given these inputs, in Listing 6.2 `schedule` gets the tag associated with `f` (Line 2) and then extracts the blocks associated with this tag in the `aapp` script (Line 3). If the follow-up strategy is different from "fail" the blocks associated with the `default` tag are appended to the list of `f`'s bocks (Line 5). Then, we obtain the list of valid workers for every block in order of appearance (Line 9). When the `workers` clause uses `*` we consider all the workers present in the configuration (Line 8). If the list of valid workers is non-empty, we choose the first one when the strategy is `best_first` (Line 12) and a random one otherwise (Line 14). If the list is empty, the schedule fails (Line 15). The `schedule` function uses the `valid` function to check when a worker is valid, i.e., it is available, it has enough capacity to host the function (Lines 18–19), and that allocating on it the function satisfies all the constraints of `capacity_used`, `max_concurrent_invocations` (Lines 21–26), and `affinity` (Lines 27–34).

Note that in aAPP the relation of (anti-)affinity is "directional"—similarly to the one introduced by Microsoft in its IaaS offering [71]. In particular, we do not impose any properties like symmetry or anti-symmetry on affinity or anti-affinity. One might argue that imposing these additional properties as well-formedness guarantees can prevent programmers from making mistakes in their aAPP scripts (e.g., they can misconfigure the policies of two functions that they wanted to be mutually anti-affine because they forgot to include a constraint in some block). While avoiding these occurrences is

```
def valid(f, w, conf, reg, block):
  (memory, tag) = reg[f]
  if (w not in conf) or (conf[w]['memory_used'] + memory > conf[w]['max_memory']):
    return False
  if 'invalidate' in block:
    if ('capacity_used' in block['invalidate']) and
          (block['invalidate']['capacity_used'] <= conf[w]['memory_used']):
        return False
    if ('max_concurrent_invocations' in block['invalidate']) and
        (block['invalidate']['max_concurrent_invocations'] <= len(conf[w]['fs'])):
      return False
  if 'affinity' in block:
    affine_tags = set([t for t in block['affinity'] if not t.startswith('!')])
    anti_affine_tags = set([t[1:] for t in block['affinity'] if t.startswith('!')])
    w_tags = set([t for (_, t) in [reg(f) for f in conf[w]['fs']]])
    for t in affine_tags:
      if t not in w_tags: return False
    for t in anti_affine_tags:
      if t in w_tags: return False
  return True
```

Listing 6.3: The pseudo-code of the `valid` function.

important, our objective is to allow aAPP to capture as many useful scenarios as possible and imposing well-formedness properties would limit the expressiveness of aAPP.[2]

We developed the code to implement the update of the functions and changed the scheduling algorithm in Scala, on a fork of the OpenWhisk repository [84]. The entire system is easily deployable using Terraform and Ansible scripts.

6.4 Performance Improvements via Affinity-awareness

To validate our platform and show that the usage of (anti-)affinity constraints for affinity-aware scenarios are beneficial, we use the example presented in Section 6.1 as a benchmark. We show that, by enforcing (anti-)affinity constraints, we can reduce average execution times and tail latency.

Recalling the example, we develop two functions, d and i, that represent a simple *divide-et-impera* serverless architecture running in a realistic co-tenancy context. Users invoke *divide* functions, requesting the solution of a problem. At invocation, *divide*

[2]For example, if we had (anti-)symmetric anti-affinity, we would not capture a scenario in which a function `init` is the seeding function for a database and function `query` manipulates that data. The function `init` should always run before `query` but never where `query` is already running, while function `query` should run where `init` is present. To obtain this behaviour, we need `init` anti-affine with `query` but `query` affine with `init`.

splits the problem into sub-problems and invokes instances of the second function, *impera*. The *impera* instances solve their relative sub-problems and store their solution fragments on a persistent storage service. After the *impera*s terminated, *divide* retrieves the partial solutions, assembles them, and returns the response to the user. Referencing the aAPP script on the right of Figure 6.1, we indicate i affine with d. In the multi-zone execution context of the use case, we have workers from two data centres (which represent the *Zone*s of Figure 6.1), placed far apart from each other. We have two synchronised instances of persistent storage (like *db* and *db*′ in Figure 6.1), one per data centre. The storage implements an eventual consistency model, i.e., it trades high availability of data off of its overall consistency. To minimise latency, both the *divide* and the *impera* functions access the storage instance closest to them. Since it can take some time for the two database instances to converge, the functions implement a traditional exponential back-off retry system—each function tries to fetch its data (sub-problems/solutions) from its local storage instance; if the data is not there, starting from a 1-second delay, the function waits for a back-off time that exponentially increases at each retry. We also draw the *heavy* functions from Figure 6.1, which simulate the possible interferences of serverless co-tenancy.

We consider three APP/aAPP scripts to showcase the benefits of (anti-)affinity constraints. The first, which uses the full expressiveness of aAPP, is the one reported on the right of Figure 6.1—where *impera*s are affine with *divide* and they are both anti-affine with the *heavy* functions. The second script removes the affinity constraints between *impera* and *divide* from the first script (anti-affinity-only-aAPP). The third script omits the anti-affinity constraints from the second one, effectively making it an APP script.

To run the use case, we deploy the OpenWhisk versions of APP and aAPP on a 8-node Kubernetes cluster on the Digital Ocean platform; one node acts as the control plane (and as such, it is unavailable to OpenWhisk), one hosts the OpenWhisk core components (i.e., the Controller, the OpenWhisk internal database CouchDB, and the messaging system Kafka), and six nodes are workers. We deploy the control plane and the OpenWhisk core components on virtual machines with 2 vCPU and 2 GB RAM, while we deploy 4 workers on virtual machines with 2 vCPU and 2 GB RAM and 2 workers with 1 vCPU and 1 GB RAM. All machines run the Ubuntu Server 20.04 OS. Location-wise, we place the control plane, the OpenWhisk core components,

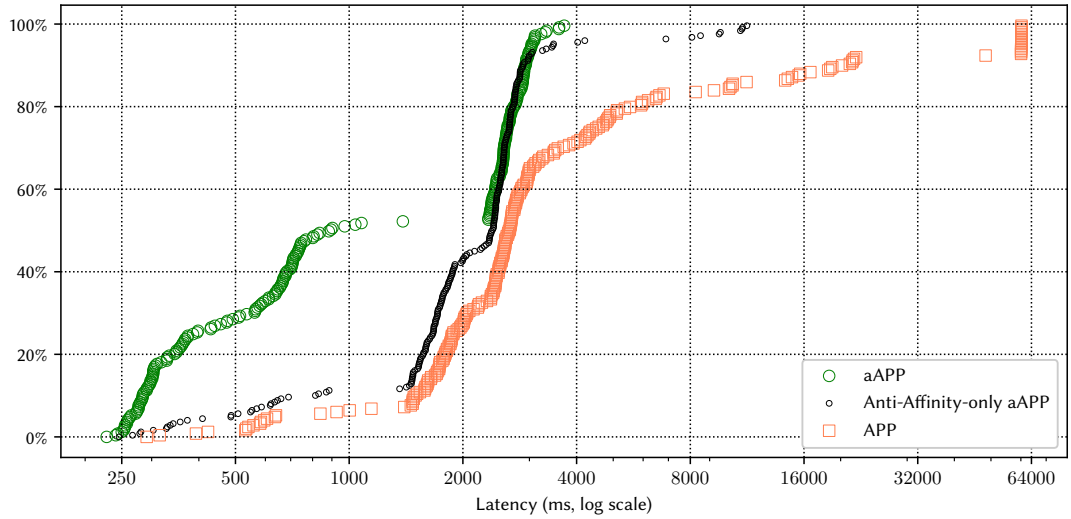


Figure 6.3: Sorted scatter plot of *divide* functions; x is the latency (ms) of the $y^{\text{th}}\%$ fastest invocation.

and 3 workers in Europe and 3 workers in North America (2 with the more powerful configuration and 1 with the lesser one in each zone). To implement persistent storage, we deploy a 2-node MongoDB replica set, one in Europe and one in North America, using the 6.0.2 version of the Community Server. We distribute the load generated by the *heavy* functions on the platform with two variants, *heavy_eu* and *heavy_us* which, in the APP/aAPP scripts, we constrain to be resp. allocated in the Europe and the North America data centres on the less powerful workers, to further amplify the effect of co-tenancy they exert. All functions are in JavaScript and run on OpenWhisk NodeJS runtime *nodejs:14*.

Experiments and Results Each experiment involves 5 sequential runs. Each run invokes the *heavy_eu* and *heavy_us* functions in non-blocking mode, followed by 10 calls of the *divide* function, each one waiting for the previous to complete. Upon termination of the *heavy* functions, we proceed with the remaining runs; for a total of 10 *heavy* and 50 *divide* functions per experiment. To ensure reliable results, we ran the experiment 5 times, totalling 250 calls of the *divide* function for each of the three APP/aAPP script. We use Apache JMeter [53] to simulate each request, tracking its latency, number of retries (to retrieve storage data), and outcomes (success or failure).

The results match our expectations. The mean and median latency for the divide functions in aAPP is resp. 1547ms and 883ms, while the 95^{th} tail latency is 3041ms. The corresponding figures increase for anti-affinity-only-aAPP: 2337ms (+40%), 2381ms (+91%), and 3476ms (+13%). As expected, the latency increases even more substantially for APP, with respectively (percentage increase vs aAPP) 8118ms (+135%), 2648ms (+99%), and 60157ms (+180%).

To further analyse the differences, in Figure 6.3, we report the plots where we sort the latencies of the *divide* functions from the shortest to the longest (x-axis). We focus on this measure because it offers a comprehensive overview of the performance of the architecture. In particular, it includes the latencies of the related *impera* functions and its latencies are concretely the ones experienced by the users interacting with the system. The first striking observation is that the distribution of the aAPP data points is interrupted (there are almost no instances) between the 1000ms and the 2400ms mark. We attribute this behaviour to having OpenWhisk core components installed in one region, which exert some overhead on the workers of the other region when they interact with the platform (e.g., to fetch functions and receive/send requests/notifications). We see similar intervals, although less apparent, for APP and anti-affinity-only-aAPP.

In the 200–1000ms interval aAPP provides consistent, fast performance, while APP and anti-affinity-only-aAPP show only a few well-performing cases—the rest, on the same performance bracket, are shifted to the right, achieving slower results. We can characterise the "fast" invocations as those where the *divide* and its two *impera* functions appear on a "free" node, i.e., without the *heavy* function, in Europe. Specifically, when using APP, each invocation has a $2/6$ probability of appearing on a free node in Europe, i.e., the probability of fast invocations is $(2/6)^3 \approx 3.7\%$; using anti-affinity-only-aAPP the figure becomes $(1/2)^3 = 12.5\%$ (each invocation has a $1/2$ chance of appearing on a European free node). Finally, using aAPP the probability raises to 50%, as all three functions go on the same node (either in the US or in the EU).

Overall, already introducing anti-affinities improves performance (mean, median, tail latency improve resp. of 110%, 10%, and 178%), which shows the impact of sharing a worker with *heavy* functions—APP shows a long tail of invocations after the ca. 3000ms mark in Figure 6.3. Looking at worst cases, using aAPP does not result in a considerable performance increase. This is visible from the plot by noticing how the tail high-percentage instances of anti-affinity-only-aAPP and aAPP almost overlap, resulting

in a small (+13%) improvement in tail latency. The differences in mean (+40%) and median (+91%) latency between having affinities or not emerges in the 250–1000ms bracket, where not having affinity causes to have only a few data points w.r.t. to the higher number of fast instances of aAPP. Practically, the figures and distribution show how strongly North American allocations impact latency vs the benefit of co-location. Besides increasing performance, aAPP succeeds in eliminating database access retries, contrarily to anti-affinity-only-aAPP (i.e., 42 requests suffer at least one retry in APP, 23 in anti-affinity-only-aAPP, and 0 in aAPP).

6.5 aAPP's Overhead is Negligible

We now show that the added functionalities (to track the state of functions on workers) of our aAPP-based prototype have negligible impact on the platform's performance.

For the experiments, we decided to use the benchmark suite used in one of our prior works [85] to benchmark their APP-based OpenWhisk implementation. Note that, in our settings, we are not interested in the data locality capabilities of APP but only in checking the scheduling performances of aAPP. Thus, we decided to deploy the platforms in only one cloud zone and use 2000 invocations for each scenario, to simplify as much as possible the testing environment and have enough invocations to draw meaningful comparisons. The benchmarks are:

- *hello-world* implements a simple echo application, and indicates the baseline performance of the platform.
- *long-running* waits for 3 seconds and benchmarks the handling of multiple functions running for several seconds and the management of their queueing process;
- *compute-intensive* multiplies two 10^2 square matrices and returns the result to the caller. This benchmark measures both the performance of handling functions performing some meaningful computation and of handling large invocation payloads.
- *DB-access (light)* executes a query for a document from a remote MongoDB database. The requested document is lightweight, corresponding to a JSON

document of 106 bytes, with little impact on computation. We used the case to measure the impact of data locality on the overall latency. Since we have all workers in the same cloud zone, we use it to measure the overhead of scheduling functions that fetch small payloads from a local database.

- *DB-access (heavy)* regards both a memory- and bandwidth-heavy data-query function. The function fetches a large document (124.38 MB) from a MongoDB database and extracts a property from the returned JSON. Similarly to the previous function, we use this one to evaluate the overhead of scheduling functions that fetch large payloads from a local database.

- *External service* benchmarks the performance invoking an external API (Slack). This function was drawn from the Wonderless dataset [34].

- *Code dependencies* is formatter that takes a JSON string and returns a plain-text one, translating the key-value pairings into Python-compatible dictionary assignments. This case was also drawn from the Wonderless dataset [34].

For completeness, we note that we omitted the *cold-start* case from [85], which is an echo application with sizable, unused dependencies. The peculiarity of the case is its 10-minute invocation pattern, used to check the performance of the platform against cold-start times (so that the platform evicts cached copies of the function, requiring costly fetch-and-startup times at any subsequent invocation). We decided to omit this benchmark since we can observe its effects with the *hello-world* and *code-dependency* cases.

We run the benchmarks on a one-zone Google Cloud cluster with four Ubuntu 20.04 virtual machines with 4 GB RAM each, one with 2 vCPU for the OpenWhisk controller and three with 1 vCPU, resp. for two workers and a MongoDB instance for the *DB-access* cases. We run 2000 function invocations for each case in batches of 4 parallel requests (500 per thread), recording both the scheduling time (the time between the arrival of a request at the controller and the issuing of the allocation) and the execution latencies. We compare aAPP, APP, and vanilla OpenWhisk. For a fair comparison, with vanilla OpenWhisk, we set the APP/aAPP configurations with a `default` policy that falls back to the vanilla scheduler.

	OpenWhisk		APP		aAPP	
	avg	st dev	avg	st dev	avg	st dev
hello-world	0.68	1.16	0.73	1.25	0.8	1.27
long-running	0.48	0.53	0.69	0.92	0.71	1.01
compute-intens.	11.57	11.92	10.17	11.67	10.01	9.66
DB-acc., light	0.65	1.31	0.85	1.62	0.83	1.31
DB-acc., heavy	0.44	0.69	0.91	1.25	1.04	1.7
external service	1.28	2.08	1.95	3.33	1.49	2.5
code dependen.	0.64	1.06	1.0	2.27	0.86	1.8

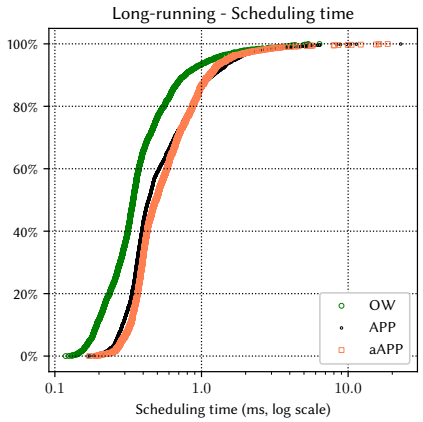


Figure 6.4: Comparison of scheduling times between vanilla, APP-, and aAPP-based OpenWhisk. From the left, avg and st dev (in ms) and the plot of the *long-running* case.

For all cases and platforms, we report on the left of Figure 6.4, in tabular form, the average (avg) and standard deviation (st dev) of the scheduling time. On average, all platforms allocate functions in less than 2ms, except for the *compute-intensive* case, which takes less than 12ms (likely due to the large request payloads that the controller needs to forward to workers). As expected, OpenWhisk vanilla is the fastest, closely (under one millisecond) followed by APP and aAPP—except for the *compute-intensive* case, where APP and aAPP perform better and OpenWhisk is slower by less than 2ms. The differences between APP and aAPP are even smaller, with APP being generally slightly (sub-millisecond) faster than aAPP. To better characterise the comparison, in Figure 6.4, we show the plot-line distribution of the scheduling times of the *long-running* case in which the average gap between aAPP and OpenWhisk is the greatest. The curve exhibit the typical tail distribution pattern [32] of cloud workloads (which accounts for the high standard deviation reported in Figure 6.4) and confirm our observations; excluding the tails, they almost overlap with negligible sub-millisecond differences.

In Figure 6.5, we report instead the latencies of execution of the cases, characterised by their average (avg), median (med), 95^{th}% tail latency (tail lat), and standard deviation (st dev), for each of the three considered platforms. Interestingly, if we consider the tail latencies, it appears that aAPP slightly outperforms OpenWhisk. We ascribe this behaviour to the high variability (as per the standard deviation in

	OpenWhisk				APP				aAPP			
	avg	med	tail lat	st dev	avg	med	tail lat	st dev	avg	med	tail lat	st dev
hello-world	88	84	126	28	78	73	114	23	76	72	109	34
long-running	3118	3096	3176	87	3094	3074	3174	86	3092	3072	3175	88
compute-intensive	348	330	559	136	304	286	501	122	257	235	409	103
DB-access, light	131	111	181	289	119	102	156	241	125	91	139	484
DB-access, heavy	95	83	130	135	95	84	131	136	87	75	113	158
external service	627	613	741	230	640	625	765	308	647	630	778	305
code depend.	132	116	213	127	143	117	255	186	98	80	142	209

Figure 6.5: Latencies of the benchmarks (in ms).

Figure 6.5) of performance of the cloud instances and the inherent variability of the cases.

6.6 Conclusions

To the best of our knowledge, aAPP is the first language that allows developers to state (anti-)affinity constraints to better control the schedule of the functions in FaaS platforms. By extending OpenWhisk, we have demonstrated the effectiveness of using (anti-)affinity constraint of aAPP in reducing latency and tail latency. Furthermore, benchmarking tests have shown that the overhead of supporting aAPP-based affinity constraints is minimal compared to vanilla OpenWhisk and its APP-based variant.

One could realise a version of aAPP for the Infrastructure and/or the Container layers, but we argue it is more interesting to focus on FaaS. There are mainstream IaaS and CaaS platforms that allow users to program directly ad-hoc schedulers (e.g., Kubernetes exposes APIs for creating scheduler plugins that define its scheduling policies). Since these layers afford a higher level of customisation than aAPP—at the expense of more technical involvement on the part of the users—a variant of aAPP for the IaaS/CaaS-levels seems less useful. On the other hand, one can use IaaS and CaaS platforms that support affinity constraints to implement affinity-aware FaaS platforms. We see two main problems with pursuing this path. The first regards performance. To implement FaaS-level affinity using IaaS/CaaS affinity constraints, we need to impose a 1:1 relation between a function instance and the VM/container running it (if we let the same VM/container run parallel copies of the same function we cannot guarantee e.g., self anti-affinity). A consequence of such an implementation is precluding the platform from exploiting the ubiquitous serverless optimisation technique of VM/container reuse

to avoid cold starts [74, 104, 102]. The second problem regards abstraction leakage, where letting FaaS users access the underlying IaaS/CaaS layers leaks details and control of the infrastructural components and breaks FaaS' paradigmatic abstractions.

Regarding the constructs we have proposed for expressing (anti-)affinity constraints in aAPP, we observe that an alternative approach could be to let the user directly declare the properties to enforce, leaving to the platform the task to realise them at run time. The scheduling runtime of this APP variant would allocate a function only if the allocation satisfies the formula or fail otherwise. The problem with this approach is scalability. Indeed, checking the satisfiability of a property's formula can take exponential time on the size of the formula, workers, and functions. Contrarily, the aAPP scheduler checks whether it can allocate a function on a worker in linear time on the size of the workers and aAPP script length.

Implementation-wise, OpenWhisk supports scenarios where multiple controllers share the pool of available workers (e.g., for redundancy and load balancing) and take scheduling decisions without coordination. In our aAPP-based implementation, such multi-controller configurations present a problem since we need to prevent scheduling races among controllers—e.g., imagine two controllers that select an available, empty worker and, at the same time, allocate mutually anti-affine functions on it. Supporting multi-controller deployments is important, but we deem dealing with it to be outside the scope of this work and an interesting subject for future work.

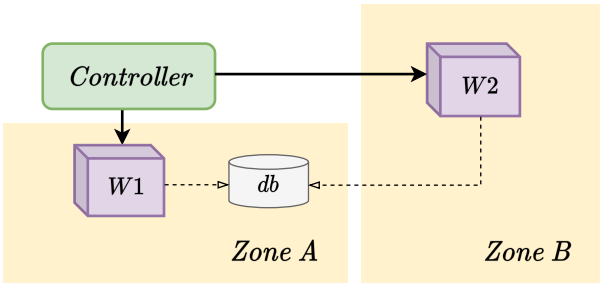
Chapter 7

Cost-aware Serverless Scheduling

7.1 Introduction

We present one last extension to the APP language we developed to investigate the integration of static analysis techniques to derive cost information from functions. Fixed and opinionated platform-wide policies to manage the allocation of function executions does not allow the platform to adapt to function performance degradation. For instance, a function can endure degradation depending on the worker that hosts it, e.g., due to effects like the latency to access data relative to the worker's location.

We visualise the issue by commenting on the minimal scenario drawn in Figure 7.1, similar to the one in Section 5.1. We have two workers, `W1` and `W2`, located in distinct geographical *Zones* A and B, respectively. Both workers can run functions that interact with a database (*db*) located in *Zone* A. When the function scheduler receives a request to execute a function, it must determine which worker to use. To minimise the function run time (and, thus, the response time), the scheduler should take into account the different computational capabilities of the workers, as well as their current workloads. Moreover, when functions interact with external services, it might take into account also their latency to access them, choosing the ones that minimise it. In our example, the scheduler should find a worker that minimises the time to access the database. In this case, that worker is `W1`, thanks to its closeness to *db* (same geographic zone) which allows it to undergo lower latencies than the farther worker `W2`. We propose to overcome the above limitations by letting users express latency-aware selection strategies. In




```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: best_first
```

Figure 7.1: A multi-zone serverless topology and APP script.

the scenario in Figure 7.1, we expect the user to be able to express policies like the following one:

```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: min_latency
```

where the strategy `min_latency` instructs the platform to give priority to the worker expected to endure the lowest latency w.r.t. its latency in the usage of external services (e.g., the database *db* in Figure 7.1).

While such high-level policies greatly alleviate the burden on users, they open a relevant question: *given a function* `f` *to be scheduled and a list of possible workers, how can one automatically guide the scheduling of* `f` *on a worker with low-latency access to* `f`*'s external services?*

We answer to the above question by proposing a solution consisting of three components:

1. the quantification of (an upper bound of) of the invocations done by a function to its external services, obtained through a static analysis of the function's code;

2. the periodical run-time monitoring of the latencies workers endure in contacting said external services;

3. the computation, at function scheduling time, of an upper-bound of the function-worker overall latency by combining the quantified invocations to the function's external services with the workers' expected latencies.

In other terms, we propose to use a combination of static analysis (applied on a function's code) and run-time monitoring (of the workers latencies in accessing the external services) to estimate a *cost* for executing a function on a worker, considering what and how it uses external services.

Thanks to such a quantification, we can support other meaningful scheduling policies like the following one:

```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
  invalidate: max_latency: 300
```

In this case, we do not specify a selection strategy (using the platform's default one) to choose between the two workers, but we consider invalid any worker whose estimated latency of running the function exceeds the threshold of 300ms.

We discuss the applicability of our approach on a minimal language, called miniSL (standing for mini Serverless Language), for programming functions in serverless applications. We focus on a minimal language for two main reasons. First, it allows us to show the feasibility of our approach by concentrating on basic language constructs, abstracting away from the specific (and, in some case, idiosyncratic) constructs of the different programming languages used in serverless computing. Second, miniSL represents an abstract language for describing the behaviour of programs written in mainstream programming languages, so that the theory developed in this chapter becomes directly

applicable to any programming language.[1] Concretely, we define a static analysis technique that, given miniSL code, extracts a set of equations that define meaningful costs, in particular, the number and kind of external service invocations. Then, we feed the equations to off-the-shelf cost analysers (e.g., PUBS [5] and CoFloCo [36]) to compute cost expressions that quantify over-approximations of said costs. The question we ask above focusses on a theoretical problem, i.e., how we can give an abstract estimation of the expected latencies of external service invocations done by a function scheduled on a given worker. We also developed a concrete proposal[2] as an extension of our in-house FaaS platform, FunLess, that we will present in the next Chapter 8. Working with a platform we have complete control over allowed us to more easily perform substantial changes to the platform's architecture to support latency monitoring, to parse the cost equations and to add a service wrapping PUBS for the analyser.

Structure of the chapter. We start, in Section 7.2, by defining our minimal language, called miniSL, which includes constructs for specifying computation flow (via `if` and `for` constructs) and for service invocation (via a `call` construct). Then, in Section 7.3, we describe how to exploit static analysis techniques, inspired by behavioural type systems like those by Garcia et al. and Laneve and Sacerdoti Coen [42, 66], to automatically extract a set of equations from function source codes written in miniSL that define meaningful function costs (in our case, the number of invocation to external services). One can feed these equations to off-the-shelf cost analyser (e.g., PUBS [5] or CoFloCo [36]) to compute cost expressions quantifying over-approximations of the considered costs. In Section 7.4, we present cAPP, our extension of APP for expressing cost-aware scheduling policies. We conclude by drawing final takeaways in Section 7.5.

7.2 The mini Serverless Language

The mini Serverless Language, shortened into miniSL, is a minimal calculus that we propose to specify the functions' behaviour in serverless computing. miniSL focuses only

[1] Since serverless platforms support many disparate programming languages, we see exploring the usage of miniSL as an abstract language for describing serverless functions too broad and tangential to be tackled here, and leave it as interesting future work.

[2] It can be found at: `https://github.com/funlessdev/funless/tree/miniSL`

on core constructs to define operations to access services, conditional behaviour with simple guards, and iterations.

Function executions are triggered by events. At triggering time, a function receives a sequence of invocation parameters: for this reason, we assume a countable set of *parameter names*, ranged over by p, p'. We also consider a countable set of *counters*, ranged over by i, j, used as indexes in iteration statements. Integer numbers are represented by n; service names are represented by `h`, `g`, ···. The syntax of miniSL is as follows (we use over-lines to denote sequences, e.g., p_1,p_2 could be an instance of $\overline{p}$):

```
F ::= (p̄) =>{S}
S ::= ε | callh(Ē)S | if (G){ S } else { S } |
      for (i in range(0,E)){ S }
G ::= E | callh(Ē)
E ::= n | i | p | E ♯ E | !E
♯ ::= + | - | * | / | > | < |
      >= | == | <= | && | ||
```

A *function* F associates to a sequence of parameters $\overline{p}$ a statement S executed at every occurrence of the triggering event. *Statements* include the empty statement ε (which is always omitted when the statement is not empty); calls to external services by means of the `call` keyword; the conditional and iteration statements. The guard of a conditional statement could be either a boolean expression or a call to an external service which, in this case, is expected to return a boolean value. The language supports standard expressions in which it is possible to use integer numbers and counters. Notice that, in our simple language, the iteration statement considers an iteration variable ranging from 0 to the value of an expression E evaluated when the first iteration starts.

In the rest of the chapter, we assume all programs to be well-formed so that all names are correctly used (e.g., counters are declared before they are used). For each expression used in the range of an iteration construct, we assume that its evaluation generates an integer, and for each service invocation `call`$\mathtt{h}(\overline{E})$, we assume that `h` is a correct service name and $\overline{E}$ is a sequence of expressions generating correct values to be passed to that service. Calls to services include serverless invocations, which possibly execute on a different worker of the caller.

We illustrate miniSL by means of three examples. As a first example, consider the code in Listing 7.1 representing the call of a function that selects a functionality based on the characteristic of the invoker.

```
1 ( isPremiumUser , par ) => {
2   if( isPremiumUser ) {
3     call PremiumService( par )
4   } else {
5     call BasicService( par )
6   }
7 }
```

Listing 7.1: Function with a conditional statement guarded by an expression.

This code may invoke either a `PremiumService` or a `BasicService` depending on whether it has been triggered by a premium user or not. The parameter `isPremiumUser` is a value indicating whether the user is a premium member (when the value is true) or not (when the value is false). The other invocation parameter `par` must be forwarded to the invoked service. For the purposes of this chapter, this example is relevant because if we want to reduce the latency of this function, the best node to schedule it could be the one that reduces the latency of the invocation of either the service `PremiumService` or the service `BasicService`, depending on whether `isPremiumUser` is true or false, respectively.

Consider now the following function, where differently from the previous version, it is necessary to call an external service to decide whether we are serving a premium or a basic user.

```
1 ( username , par ) => {
2   if( call IsPremiumUser(username) ) {
3     call PremiumService( par )
4   } else {
5     call BasicService( par )
6   }
7 }
```

Listing 7.2: Function with a conditional statement guarded by an invocation to external service.

In this case, the first parameter carries an attribute of the user (its name) but it does not indicate (with a boolean value) whether it is a premium user or not. Instead,

the necessary boolean value is returned by the external service `IsPremiumUser` that checks the username and returns true only if that username corresponds to that of a premium user. Within this setting is difficult to predict the best worker to execute such a function, because the branch that will be selected is not known at function scheduling time. If the user triggering the event is a premium member, the expected execution time of the function is the sum of the latencies of the service invocations of `IsPremiumUser` and `PremiumService` while, if the user is not a premium member, the expected execution time is the sum of the latencies of the services `IsPremiumUser` and `BasicService`. As an (over-)approximation of the expected delay, we could consider the worst execution time, i.e., the sum of the latency of the service `IsPremiumUser` plus the maximum between the latencies of the services `PremiumService` and `BasicService`. At scheduling time, we could select the best worker as the one giving the best guarantees in the worst case, e.g., the one with the best over-approximation.
Consider now a function triggering a sequence of map-reduce jobs.

```
1 ( jobs, m, r ) => {
2   for(i in range(0, m)) {
3     call Map(jobs, i)
4     for(j in range(0, r)) {
5       call Reduce(jobs, i, j)
6     }
7   }
8 }
```

Listing 7.3: Function implementing a map-reduce logic.

The parameter `jobs` describes a sequence of map-reduce jobs. The number of jobs is indicated by the parameter `m`. The "map" phase, which generates `m` "reduce" subtasks, is implemented by an external service `Map` that receives the `jobs` and the specific index `i` of the job to be mapped. The "reduce" subtasks are implemented by an external service `Reduce` that receives the `jobs`, the specific index `i` of the job under execution, and the specific index `j` of the "reduce" subtask to be executed — for every `i`, there are `r` such subtasks. In this case, the expected latency of the entire function is given by the sum of `m` times the latency of the service `Map` and of `m` $\times$ `r` times the latency of the service `Reduce`. Given that such latency could be high, a user could be interested to run the function on a worker, only if the expected overall latency is below a given threshold.

7.3 The Inference of Cost Expressions

In this section, we formalise the inference of a cost program from miniSL code. Once inferred, we can feed this program to off-the-shelf tools, such as [36, 5], to calculate the cost expression of the related miniSL code. Notice that, since these tools are designed to handle only Presburger arithmetic, we restrict our extraction only to a subset of miniSL, where the expressions conform to Presburger arithmetic constraints.
Cost programs are lists of *equations* which are terms

$$f(\overline{x}) = \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \qquad [\varphi]$$

where variables occurring in the right-hand side and in φ are a subset of $\overline{x}$ and f and f_i are (cost) function symbols. Every function definition has a right-hand side consisting of

- a *Presburger arithmetic expression* $\mathbb{e}$ whose syntax is

$$\begin{array}{rcl} \mathbb{e} ::= & x \mid q \mid \mathbb{e}+\mathbb{e} \mid \mathbb{e}-\mathbb{e} \mid q*\mathbb{e} \\ \mid & max(\mathbb{e}_1,\cdots,\mathbb{e}_k) \end{array}$$

  where x is a variable and q is a positive rational number,

- a number of *cost function invocations* $f_i(\overline{\mathbb{e}_i})$ where $\overline{\mathbb{e}_i}$ are Presburger arithmetic expressions,

- the *Presburger guard* φ is a *linear conjunctive constraint*, *i.e.*, a conjunction of *constraints* of the form $\mathbb{e}_1 \geq \mathbb{e}_2$ or $\mathbb{e}_1 = \mathbb{e}_2$, where both $\mathbb{e}_1$ and $\mathbb{e}_2$ are Presburger arithmetic expressions.

The intended meaning of an equation

$$f(\overline{x}) = \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \quad [\varphi]$$

is that the cost of f is given by $\mathbb{e}$ and the costs of $f_i(\overline{\mathbb{e}_i})$, when the guard φ is true. Intuitively, $\mathbb{e}$ quantifies the specific cost of one execution of f without taking into account invocations of either auxiliary functions or recursive calls. Such additional

cost is quantified by $\sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i})$. The *solution of a cost program* is an expression, quantifying the cost of the function symbol in the first equation in the list, which is parametric in the formal parameters of the function symbol.

For example, the following cost program

$$\begin{array}{lcll} f(N,M) & = & M+f(N-1,M) & [N\geq 1] \\ f(N,M) & = & 0 & [N=0] \end{array}$$

defines a function f that is invoked $N+1$ times and each invocation, excluding the last having cost 0, costs M. The solution of this cost program is the *cost expression* $N\times M$.

Our technique associates cost programs to miniSL functions following a syntax-directed approach: we define a set of (inference) rules that, following the parse tree bottom-up, gather fragments of cost programs that are then combined in a syntax-directed manner.

$$[\text{EPS}] \quad \Gamma \vdash \varepsilon : 0 \ ; \ \emptyset \ ; \ \emptyset \qquad\qquad [\text{CALL}] \quad \frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \ ; \ \mathsf{C} \ ; \ \mathsf{Q}}{\Gamma \vdash \mathtt{call}\,\mathtt{h}(\overline{\mathsf{E}})\,\mathsf{S} : \mathbb{e}+\mathbb{e}' \ ; \ \mathsf{C} \ ; \ \mathsf{Q}}$$

$$[\text{IF-EXP}] \quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{E} : \varphi \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \ ; \ \mathsf{C} \ ; \ \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \ ; \ \mathsf{C}' \ ; \ \mathsf{Q}' \qquad \mathit{if}_\ell \ \mathit{fresh} \\ \overline{w} = \mathit{var}(\varphi,\mathbb{e}',\mathbb{e}'') \cup \mathit{var}(\mathsf{C},\mathsf{C}') \qquad \mathsf{Q}'' = \left[\begin{array}{ll} \mathit{if}_\ell(\overline{w}) = \mathbb{e}'+\mathsf{C} & [\ \varphi\ ] \\ \mathit{if}_\ell(\overline{w}) = \mathbb{e}''+\mathsf{C}' & [\neg\varphi] \end{array}\right]\end{array}}{\Gamma \vdash \mathtt{if}\ (\mathsf{E})\{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} : 0 \ ; \ \mathit{if}_\ell(\overline{w}) \ ; \ \mathsf{Q},\mathsf{Q}',\mathsf{Q}''}$$

$$[\text{IF-CALL}] \quad \frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \ ; \ \mathsf{C} \ ; \ \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \ ; \ \mathsf{C}' \ ; \ \mathsf{Q}'}{\Gamma \vdash \mathtt{if}\ (\mathtt{call}\,\mathtt{h}(\overline{\mathsf{E}}))\{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} : \mathbb{e}+\mathit{max}(\mathbb{e}',\mathbb{e}'') \ ; \ \mathsf{C}+\mathsf{C}' \ ; \ \mathsf{Q},\mathsf{Q}'}$$

$$[\text{FOR}] \quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{E} : \mathbb{e} \qquad \Gamma + i : \mathtt{Int} \vdash \mathsf{S} : \mathbb{e}' \ ; \ \mathsf{C} \ ; \ \mathsf{Q} \qquad \overline{w} = (\mathit{var}(\mathbb{e},\mathbb{e}') \cup \mathit{var}(\mathsf{C})) \setminus i \\ \mathit{for}_\ell \ \mathit{fresh} \qquad \mathsf{Q}' = \left[\begin{array}{ll} \mathit{for}_\ell(i,\overline{w}) = \mathbb{e}'+\mathsf{C}+\mathit{for}_\ell(i+1,\overline{w}) & [\ \mathbb{e} \geq i\ ] \\ \mathit{for}_\ell(i,\overline{w}) = 0 & [\ i \geq \mathbb{e}+1\ ] \end{array}\right]\end{array}}{\Gamma \vdash \mathtt{for}\ (i\ \mathtt{in}\ \mathtt{range}(\mathtt{0},\mathsf{E}))\{\ \mathsf{S}\ \} : 0 \ ; \ \mathit{for}_\ell(0,\overline{w}) \ ; \ \mathsf{Q},\mathsf{Q}'}$$

$$[\text{PRG}] \quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{S} : \mathbb{e} \ ; \ \mathsf{C} \ ; \ \mathsf{Q} \qquad \overline{w} = \mathit{var}(\overline{p},\mathbb{e}) \cup \mathit{var}(\mathsf{C}) \\ \mathit{main}\ \mathit{fresh} \qquad \mathsf{Q}' = \mathit{main}(\overline{w}) = \mathbb{e}+\mathsf{C} \quad [\,]\end{array}}{\Gamma \vdash (\overline{p})\ \mathtt{=>}\{\mathsf{S}\} : \mathsf{Q}',\mathsf{Q}}$$

Figure 7.2: The rules for deriving cost expressions

As usual with syntax-directed rules, we use *environments* Γ, Γ', which are maps. In particular,

- Γ takes a service $\mathtt{h}$ or a parameter name p and returns a Presburger arithmetics expression, which is usually a variable. For example, if $\Gamma(\mathtt{h}) = X$, then X will appear in the cost expressions of miniSL functions using $\mathtt{h}$ and will represent the cost for accessing the service. As regards parameter names p, $\Gamma(p)$ represents values which are known at function scheduling time,

- Γ takes counters i and returns the type `Int`.

When we write $\Gamma{+}i{:}\mathtt{Int}$, we assume that i does not belong to the domain of Γ. Let C be a sum of (cost) function invocations and let Q be a list of equations. Judgments have the shape

- $\Gamma \vdash \mathsf{E}{:}\mathbb{e}$, meaning that the value of the *integer expression* E in Γ is represented by (the Presburger arithmetic expression) $\mathbb{e}$,
- $\Gamma \vdash \mathsf{E}{:}\varphi$, meaning that the value of the *boolean expression* E in Γ is represented by (the Presburger guard) φ,
- $\Gamma \vdash \mathsf{S}{:}\mathbb{e}\ ;\ \mathsf{C}\ ;\ \mathsf{Q}$, meaning that the cost of S in the environment Γ is $\mathbb{e}{+}\mathsf{C}$ given a list Q of equations,
- $\Gamma \vdash \mathsf{F}{:}\mathsf{Q}$, meaning that the cost of a miniSL function F in the environment Γ is given by the cost program Q (remember that a cost program is a list of equations).

We use the notation $var(\mathbb{e})$ to address the set of variables occurring in $\mathbb{e}$, which is extended to tuples $var(\mathbb{e}_1,\cdots,\mathbb{e}_n)$ with the standard meaning. Similarly $var(\sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i}))$ is the union of the sets of variables $var(\overline{\mathbb{e}_0}),\cdots,var(\overline{\mathbb{e}_n})$. We use $var(\varphi)$ for Presburger guards.

The inference rules for miniSL are reported in Figure 7.2. They compute the cost of a program with respect to the calls to external services (whose cost is recorded in the environment Γ). Therefore, if a miniSL expression (or statement) has no service invocation, its cost is 0. Notice that in the rule [IF-EXP] we use the guard $[\neg\varphi]$, to model the negation of a linear conjunctive constraint φ, even if negation is not permitted in Presburger arithmetic. Actually, such notation is syntactic sugar defined as follows:

- let $\neg\varphi$ (the *negation* of a Presburger guard φ) be the *list* of Presburger guards

$$\begin{aligned}\neg(\mathbb{e}{\geq}\mathbb{e}') &= \ \mathbb{e}'{\geq}\mathbb{e}{+}1\\ \neg(\mathbb{e}{=}\mathbb{e}') &= \ \mathbb{e}{\geq}\mathbb{e}'{+}1\ ;\ \mathbb{e}'{\geq}\mathbb{e}{+}1\\ \neg(\mathbb{e}{\wedge}\mathbb{e}') &= \ \neg\mathbb{e}\ ;\ \neg\mathbb{e}'\end{aligned}$$

  where ; is the list concatenation operator (the list represents a *disjunction of Presburger guards*),

- let $\neg\varphi = \varphi_1 \ ; \ \cdots \ ; \ \varphi_m$, where φ_i are Presburger guards, then

$$\begin{aligned}&\left(f(\overline{x}) = \mathbb{e} + \textstyle\sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i})\right)[\neg\varphi]\\ &\quad \stackrel{\text{def}}{=} \left\{ f(\overline{x}) = \mathbb{e} + \textstyle\sum_{i\in 0..n} f_i(\overline{\mathbb{e}_i}) \quad [\varphi_j] \quad \mid \quad j \in 1..m \right\}.\end{aligned}$$

We now comment on the inference rules reported in Figure 7.2.[3]

Rule [CALL] manages invocation of services: the cost of $\texttt{call}\,\mathtt{h}(E)\,\mathsf{S}$ is the cost of S plus the cost for accessing the service $\mathtt{h}$.

Rule [IF-EXP] defines the cost of conditionals when the guard is a Presburger arithmetic expression that can be evaluated at function scheduling time. We use a corresponding cost function, if_ℓ, whose name is fresh,[4] to indicate that the cost of the entire conditional statement is either the cost of the then-branch or the else-branch, depending on whether the guard is true or false. As discussed above, the use of the guard $\neg\varphi$ generates a list of equations.

Rule [IF-CALL] defines an upper bound of the cost of conditionals when the guard is an invocation to a service. At scheduling time it is not possible to determine whether the guard is true or false – *c.f.* the second example in Section 7.2. Therefore the cost of a conditional is the maximum between the cost $\mathbb{e}' + \mathsf{C}$ of the then-branch and the one $\mathbb{e}'' + \mathsf{C}'$ of the else-branch, plus the cost $\mathbb{e}$ to access to the service in the guard. However, considering that the expression $\mathit{max}(\mathbb{e}+\mathsf{C},\mathbb{e}'+\mathsf{C}')$ is not a valid right-hand side for the equations in our cost programs, we take as over-approximation the expression $\mathit{max}(\mathbb{e},\mathbb{e}')+\mathsf{C}+\mathsf{C}'$.

As regards iterations, according to [FOR], its cost is the invocation of the corresponding function, for_ℓ, whose name is fresh (we assume that iterations have pairwise different line-codes). The rule adds the counter i to Γ (please recall that $\Gamma + i : \mathtt{Int}$ entails that $i \notin \mathit{dom}(\Gamma)$). In particular, the counter i is the first formal parameter of for_ℓ; the other parameters are all the variables in $\mathbb{e}$, in notation $\mathit{var}(\mathbb{e})$ plus those in the invocations C (minus the i). There are two equations for every iteration: one is the case when i is out-of-range, hence the cost is 0, the other is when it is in range and

[3] We omit rules for expressions E since they are straightforward: they simply return E if E is in Presburger arithmetics. We notice that no rule is defined if E *is not* in Presburger arithmetics. In fact, in these cases, it is not possible to derive cost equations.

[4] We assume that conditionals have pairwise different line-codes and ℓ represents the line-code of the if in the source code.

the cost is the one of the body *plus* the cost of the recursive invocation of for_ℓ with i increased by 1.

The cost of a miniSL program is defined by [PRG]. This rule defines an equation for the function *main* and puts this equation as the first one in the list of equations [5]. Once inferred, we can feed this program to off-the-shelf tools, such as [36, 5], which will compute the cost of the the *first function* of the list, *i.e.* the *main* function.

As an example, we apply the rules of Figure 7.2 to the codes in Listings 7.1, 7.2, and 7.3. Let $\Gamma(\texttt{isPremiumUser}) = u$, $\Gamma(\texttt{par}) = v$, $\Gamma(\texttt{PremiumService}) = P$ and $\Gamma(\texttt{BasicService}) = B$. For Listing 7.1, we obtain the cost program

$$\begin{array}{rll} main(u,v,P,B) = & if_2(u,P,B) & [\,] \\ if_2(u,P,B) = & P & [\,u=1\,] \\ if_2(u,P,B) = & B & [\,u=0\,] \end{array}$$

Notice that the parameters of the *main* function include, initially, the values corresponding to the parameters of the corresponding miniSL function and then those corresponding to the other variables occurring in the cost equations.
For Listing 7.2, let $\Gamma(\texttt{username}) = u$, $\Gamma(\texttt{par}) = v$,
$\Gamma(\texttt{IsPremiumUser}) = K$, $\Gamma(\texttt{PremiumService}) = P$ and $\Gamma(\texttt{BasicService}) = B$. Then the rules of Figure 7.2 return the single equation

$$main(u,v,K,P,B) = \quad K + max(P,B) \qquad [\,]$$

For 7.3, when $\Gamma(\texttt{jobs}) = J$. $\Gamma(\texttt{m}) = m$, $\Gamma(\texttt{r}) = r$, $\Gamma(\texttt{Map}) = M$ and $\Gamma(\texttt{Reduce}) = R$, the cost program is

$$\begin{array}{rll} main(J,m,r,M,R) = & for_2(0,m,r,M,R) & [\,] \\ for_2(i,m,r,M,R) = & M + for_4(0,r,R) + & \\ & for_2(i+1,m,r,M,R) & [\,m \geq i\,] \\ for_2(i,m,r,M,R) = & 0 & [\,i \geq m+1\,] \\ for_4(j,r,R) = & R + for_4(j+1,r,R) & [\,r \geq j\,] \\ for_4(j,r,R) = & 0 & [\,j \geq r+1\,] \end{array}$$

[5] Given that miniSL functions are anonymous, we use the default name *main* for the corresponding cost function.

The foregoing cost programs can be fed to automatic solvers such as PUBS [5] and CoFloCo [36]. The evaluation of the cost program for Listing 7.1 returns $max(P,B)$ because u is unknown. On the contrary, if u is known, it is possible to obtain a more precise evaluation from the solver: if $u=1$ it is possible to ask the solver to consider $main(1,v,P,B)$ and the solution will be P, while if $u=0$ it is possible to ask the solver to consider $main(0,v,P,B)$ and the solution will be B. The evaluation of $main(u,v,K,P,B)$ for Listing 7.2 gives the expression $K+max(P,B)$, which is exactly what is written in the equation. This is reasonable because, statically, we are not aware of the value returned by the invocation of `IsPremiumService`. Last, the evaluation of the cost program for Listing 7.3 returns the expression $m\times(M+r\times R)$.

Since we combine miniSL and our inference system for estimating costs of functions interacting with external services, one might wonder how relevant the approach is, i.e., how common are serverless functions that call external services, and what is their structure? While a systematic study is out of the scope of this thesis, we started this process by analysing a comprehensive repository of illustrative serverless functions[6] for different platforms (AWS, Azure, OpenWhisk, etc.). Our analysis reveals that 50% (65/130) of these functions follow patterns that one can represent using miniSL by abstracting away structured data and internal computation and estimate their cost w.r.t. the flow of external calls, such as HTTP invocations to external services.

7.4 From APP to cAPP

We now present the new language cAPP for expressing cost-aware function scheduling policies, by extending the previously discussed language APP, as shown in Figure 7.3.

7.4.1 Cost-aware policies with cAPP

To support the scheduling of functions based on costs we propose two extensions to APP. The first one is a new selection strategy named `min_latency`. Such a strategy selects, among some available workers, the one which minimises a given cost expression. The second one is a new invalidation condition named `max_latency`. This condition

[6] "A collection of ready-to-deploy Serverless Framework services" at `https://github.com/serverless/examples`.

policy_tag ∈ *Identifiers* ∪ {`default`} *label* ∈ *Identifiers* $n \in \mathbb{N}$

```
app       ::= - tag
tag       ::= policy_tag : - block  followup?
block     ::= workers: [ * | - wrk : worker_label ]
              (strategy: [ random | platform | best_first
                         | min_latency ])?
              (invalidate: [ capacity_used : n%
                           | max_concurrent_invocations: n
                           | overload
                           | max_latency: n
                           ])?
)
followup  ::= followup: [ default | fail ] )
```

Figure 7.3: The syntax of cAPP (the extensions from APP are highlighted).

invalidates a worker in case the corresponding cost expression is greater than a given threshold.

We dub cAPP the cost-aware extension of APP and illustrate its main features by showing examples of cAPP scripts that target the functions in Listings 7.1–7.3.

```
- premUser:
 - workers:
     - wrk: W1
     - wrk: W2
   strategy: min_latency
```

Listing 7.4: cAPP script for Listings 7.1 and 7.2.

Listing 7.4 defines a cAPP tagged `premUser` that we will associate to both the functions at Listing 7.1 and 7.2. In this script, we specify to follow the logic `min_latency` to select among the two workers, `W1` and `W2` listed in the `workers` clause, and prioritises the one for which the solution of the cost expression is minimal.

To better illustrate the phases of the `min_latency` strategy, we depict in Figure 7.4 the flow, from the deployment of the cAPP script to the scheduling of the functions in

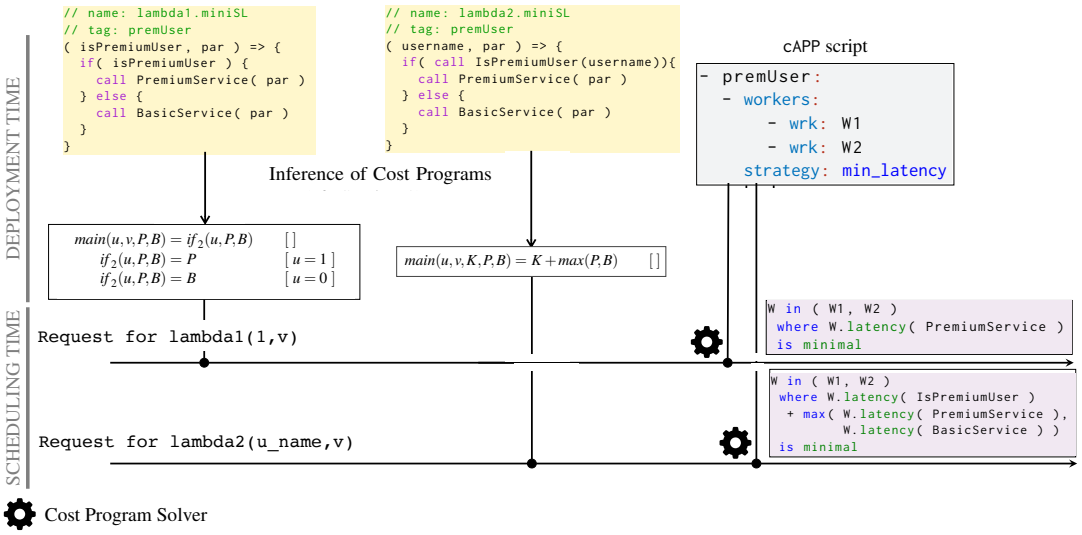


Figure 7.4: Flow followed, from deployment to scheduling, of the functions at Listings 7.1 and 7.2.

Listings 7.1 and 7.2. When the cAPP script is created, the association between the functions code and their cAPP script is specified by tagging the two functions with the comment `// tag:premUser`. In this phase, assuming the scheduling policy of the cAPP script requires the computation of the functions cost (because the strategy is `min_latency`), the code of the functions is used to infer the corresponding cost program. When the functions are invoked, i.e., at scheduling time, we can compute the solution of the cost program, given the knowledge of the invocation parameters. The knowledge of the invocation parameters allows for a more precise analysis. For instance, for the function in Listings 7.1, called `lambda1`, it is possible to invoke the cost analyser with either $main(1,v,P,B)$ or $main(0,v,P,B)$ where P represent the cost of `PremiumService`, B the cost of `BasicService` and the first parameter is the value of the `isPremiumUser` parameter.

If the invocation is `lambda1`$(1,v)$ (first horizontal line in In Figure 7.4) then the cost program (represented by the intersection point on the left) and the corresponding cAPP policy to implement the expected scheduling policy are retrieved. At this point, a cost analyser is used to solve the cost programs (depicted by the gear). In this case, since the cost expression is P, which is `PremiumService`, the scheduling amounts to (i) estimating the latencies to access to `PremiumService` from the considered workers

and (*ii*) choosing the worker that minimises the foregoing latency. This computation is highlighted in the rightmost grey window corresponding to the request `lambda1`(1,v).

When the request is `lambda2`(u_name, v), the corresponding cost function is $main(u\_name, v, K, P, B)$, where K is the cost of the service `IsPremiumUser`. In this case, the cost expression is $K + max(P,B)$ Since `lambda2.miniSL` has the same tag as `lambda1.miniSL`, the selected `cAPP` script is the same. Therefore the scheduling amounts to minimize the latencies from the workers `W1` and `W2` to the services `IsPremiumUser`, `PremiumService` and `BasicService` according to the expression $K + max(P, B)$. This is highlighted in the rightmost grey window corresponding to the request `lambda2`(u_name,v).

The controller needs also to be aware of the possibility of invalidating a worker when the latency to access a service exceeds a certain threshold. In particular, when `max_latency` is used in the `invalidate` clause, workers are not selected if the computed latency is above the given value. To illustrate this item, let us consider the `cAPP` code for the map-reduce function in Listing 7.5.

```
- mapReduce:
  - workers:
    - wrk: W1
    - wrk: W2
  strategy: random
  invalidate:
    max_latency: 300
```

Listing 7.5: `cAPP` script for Listing 7.3.

As visualised in Figure 7.5, starting from the (top-most) deployment phase box where we tag the function (`//tag:mapReduce`), the cost program is computed, obtaining the associated cost expression. Then, when a request for the function is received, the execution of the `cAPP` policy is triggered, which selects one of the two workers `W1` or `W2` at `random` and checks their validity following the logic shown at the bottom of Figure 7.5, i.e., the cost program is solved and the parameters `m` and `r` are replaced with the `latency` to contact the `Map` and `Reduce` services from the selected worker, and possibly invalidate it if the computed value is greater than `300`.

```
// tag: mapReduce
( jobs, m, r ) => {
  for(i in range(0, m)) {
    call Map(jobs, i)
    for(j in range(0, r)) {
      call Reduce(jobs, i, j)
    }
  }
}
```

$$\Downarrow$$

$$
\begin{array}{lll}
main(J,m,r,M,R) = & for\_2(0,m,r,M,R) & [\,] \\
for_2(i,m,r,M,R) = & M + for_4(0,r,R) + for_2(i+1,m,r,M,R) & [\, m \geq i \,] \\
for_2(i,m,r,M,R) = & 0 & [\, i \geq m+1 \,] \\
for_4(j,r,R) = & R + for_4(j+1,r,R) & [\, r \geq j \,] \\
for_4(j,r,R) = & 0 & [\, j \geq r+1 \,]
\end{array}
$$

$$\Downarrow$$

```
Cost Expression: m*(M + r*R)
```

$$\Downarrow$$

```
W in ( W1, W2 )
 where m *( W.latency( Map )
          + r * W.latency( Reduce ) )
 is < 300
```

Figure 7.5: The map-reduce function, its cost analysis, and scheduling invalidation logic.

7.5 Conclusions

We introduced a framework that lightens the burden on the shoulders of users by deriving cost information from the functions, via static analysis, into a cost-aware variant of APP that we call cAPP. To show the feasibility of the approach, we present a prototype of such framework where we extract cost equations from functions' code, synthesise cost expressions through off-the-shelf solvers, and implement cAPP to support the specification of cost-aware allocation policies. Specifically, we demonstrate that one can over-approximate, at scheduling time, the overall latency endured by the invocation of a function `f` when running on a given worker and use this information to govern its

scheduling. To achieve this result, we present a proposal for an extension of the APP language, called cAPP, to make function scheduling cost-aware. The extension adds new syntactic fragments to APP so that programmers can govern the scheduling of functions towards those execution nodes that minimise their calculated latency (e.g., increasing serverless function performance) and avoids running functions on nodes whose execution time would exceed a maximal response time defined by the user (e.g., enforcing quality-of-service constraints).

In future work, we will address several key questions that remain open. Specifically, we aim to investigate the scalability and performance of our approach by examining how it would work with more complex examples and evaluating its execution times under varied computational conditions. Since determining the exact cost of a function is, in principle, undecidable, as future work, we will focus on exploring models and techniques that can make this problem more tractable in practical scenarios. This may include the development of heuristics and over-approximation methods that work effectively for the majority of cases, while ensuring that these approaches remain computationally efficient. Additionally, we are considering architectural solutions to complement these techniques, such as the inclusion of caching systems to store and reuse previously computed costs for repeated function invocations. These systems could significantly reduce overhead by calculating the actual cost of a function only once, avoiding redundant computations. To further enhance system reliability, we propose integrating timeouts for particularly challenging cost calculations, paired with sensible default strategies to maintain responsiveness. This would ensure the system remains functional even in scenarios where exact costs cannot be computed within a reasonable time frame.

Chapter 8

FunLess: Lightweight Cloud-Edge FaaS

8.1 Introduction

While public clouds are the birthplace of serverless computing, recent industrial and academic proposals demonstrated the desirability, benefits and feasibility of moving FaaS outside public clouds. These solutions are tailored for private, public, and hybrid (where the infrastructure includes parts from public and private) cloud scenarios [24] and include edge [16] and Internet-of-Things [96] components. From the industrial point of view, several FaaS platforms are designed for edge computing (e.g., AWS Greengrass[1], Cloudfare Workers[2]).

In contrast to public edge-cloud computing solutions, *private edge cloud systems* have the benefit of further reducing latency, increasing security and privacy, and improving bandwidth and usage of high-end devices [96]. More precisely, private edge cloud systems are small-scale cloud data centres in a local physical area, such as a house, an office, a factory, or a small geographic area, where mobile devices, such as drones, mobile robots, smartphones and fixed devices, such as sensors/actuators, workstations, and servers are interconnected through sisngle or multiple local area networks.

In this chapter, we address the challenge of supporting FaaS in private edge cloud

[1] `https://aws.amazon.com/greengrass/`.

[2] `https://www.cloudflare.com/en-gb/learning/serverless/glossary/serverless-and-cloudflare-workers/`.

systems. Off-the-shelf solutions to this challenge consist of deploying popular open-source FaaS platforms (e.g., OpenFaas, Knative, Fission, OpenWhisk) on top of container orchestration technologies (e.g., Kubernetes). However, these technologies, which usually rely on containers and container orchestration solutions, entail performance and resource overheads which can create issues on devices with constrained resources—they might not have enough memory to host containers or computational power to effectively run functions, especially in low-latency application contexts.

These problems motivated researchers and practitioners to consider alternatives and propose runtimes that provide the isolation and parallel execution of existing FaaS platforms yet mediate the heavy toll of the mentioned more complex runtimes. Examples of these proposals include using virtual machines like that of Java [89] and Python [43] or embedding functions in unikernels [73]. Unfortunately, while these solutions achieve the goal of reducing the overhead of containers, they respectively miss fundamental features. Java/Python VMs do not provide high-performing runtimes [51] and properly isolate functions (e.g., exposing the users to security risks). Unikernels are still a niche technology whose usage requires specific engineering knowledge (e.g., to define the minimal OS stack needed to run high-level functions).

A promising alternative is WebAssembly[3] (Wasm) for lightweight FaaS environments [57] (introduced in more detail in Section 8.2). Indeed, Wasm comes with a stack-based virtual machine designed for running programs in a sandbox environment with performance close to native code and fast load times. Wasm proved to be a valid candidate for FaaS, providing lightweight sandboxing at the edge with both small latencies and startup times [45, 41]—recently, providers like Cloudflare proposed closed-source solutions based on Wasm[4].

FunLess. Building on these results, we propose FunLess, a FaaS platform designed for (mixed) edge-cloud scenarios. FunLess uses Wasm to run functions, providing many pros:

- *Security.* Wasm's inherent security and isolation mechanisms make it well-suited for scenarios where data integrity and confidentiality are critical.

- *Memory and CPU footprint.* FunLess does not require a container runtime (e.g.,

[3] `https://webassembly.org/`.

[4] `https://developers.cloudflare.com/workers/runtime-apis/webassembly/`.

Docker) and orchestrator (e.g., Kubernetes). Hence, the "bare-metal" deployment of FunLess frees resources essential for running functions on memory-constrained or low-power edge devices.

- *Cold starts.* FunLess leverages Wasm to mitigate the problem of cold starts [113], i.e., delays in function execution due to the overhead of loading and initialising functions—an issue that constrained-resource edge devices can accentuate. Cold-start mitigations usually rely on caching or keeping "warm" function instances. However, the size of containers can make these solutions unfeasible on constrained-resource devices. FunLess's use of Wasm minimise the cost of function caching (and even fetch-and-load roundtrips), making cold-start mitigations more affordable. Moreover, Wasm runtimes provide fast startup times (Wasm's main use case is in-browser execution, where responsiveness is crucial), allowing FunLess to achieve small cold-start overheads.

- *Consistent function development and deployment environment.* Since Wasm abstracts away the hardware and environment it runs within, FunLess provides a consistent development and deployment experience across the diverse private edge architectures, offering a built-in solution for the challenges of variability in hardware and software environments of private edge-cloud scenarios. Similarly to Java bytecode, Wasm binaries can run on any platform that can execute a (dedicated) Wasm runtime. As illustrated in Section 8.3, the developers only need to write once their functions[5], compile them into Wasm binaries, and load them into the platform. FunLess handles the task of running them on the possible diverse devices and architectures of the given cloud/edge infrastructure.

- *Simple and flexible platform deployments.* FunLess can use existing containerisation solutions (e.g., Kubernetes) to streamline and ease its deployment. When container orchestration technologies are not affordable/available, users can install FunLess by running a *Core* component (with metrics and storage services, e.g., resp. Prometheus and Postgres) on a node and a *Worker* component on the nodes tasked to run the functions (cf. Section 8.3). This flexibility derives from

[5]FunLess users can write functions in any language supported by the platform, currently Rust, Go, and JavaScript.

WebAssembly (the binaries do not need an ulterior container for their isolation), and FunLess' communication mechanism between nodes.

Structure of the chapter In the following, in Section 8.2, we present WebAssembly in more detail. We detail FunLess' architecture in Section 8.3 and show an analysis of the energy consumption of FunLess compared to OpenWhisk and a classical container-based service architecture in Section 8.4. We then draw our conclusions and future work directions in Section 8.5.

8.2 WebAssembly

We dedicate this section to providing the preliminary notions useful to contextualise our contribution. Specifically, we introduce WebAssembly—the technology underpinning the FunLess execution runtime (cf. Section 8.3). The WebAssembly [122] technology, Wasm for short, is a W3C standard since 2019, maintained with contributions from Apple, Google, Microsoft, Mozilla, and other companies. The idea behind Wasm is to provide a simple assembly-like instruction set which can run efficiently within a browser. At its core, Wasm includes a binary instruction format and a stack-based virtual machine that supports functions and control flow abstractions like loops and conditionals. Although browsers are the main target of Wasm, recent initiatives, like WebAssembly System Interface [123] (WASI), norm the implementation of Wasm runtimes to support the execution of Wasm code outside the browser with a set of APIs that provide POSIX capabilities (e.g., file system, network, and process management). Some examples of open-source and proprietary WASI-compliant runtimes are Wasmtime [121], Wasmer [120], and WasmEdge [119].

Focusing on FaaS, Wasm provides a sandboxed runtime environment for functions, akin to containers. However, while one needs to build a container (for the same function) for each targeted architecture, the same Wasm binary can run on different architectures thanks to the hardware abstraction provided by the Wasm runtime. Moreover, Wasm binaries tend to be more lightweight than containers, thanks to the fact that they do not need to include a pre-packaged filesystem.

8.3 Platform Architecture

We present the principles and technologies behind FunLess, its architecture and discuss our design choices (trade-offs and limitations).

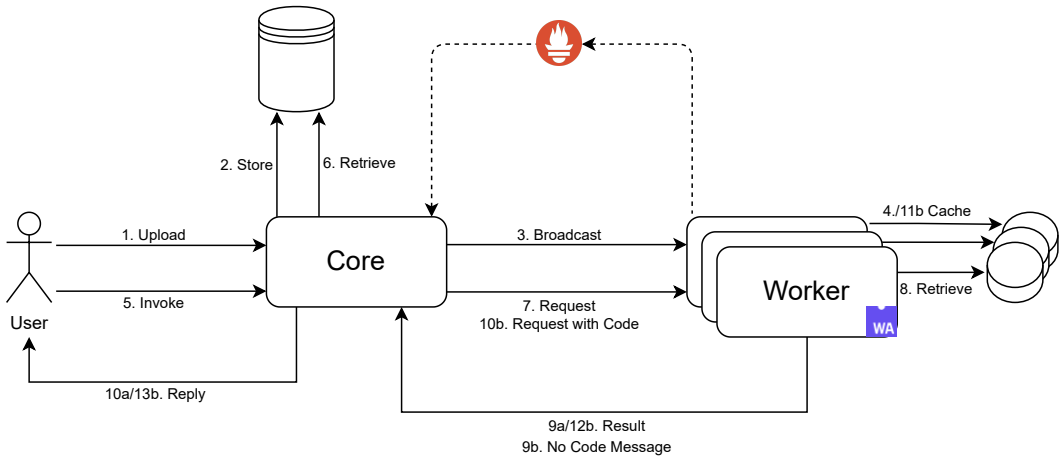


Figure 8.1: Architecture of the FunLess platform with the function flow from creation to invocation.

The main principles behind the design of FunLess are the simplicity of both function development and platform deployment and the flexibility of hardware and deployment automation. In particular, FunLess is independent of the underlying deployment orchestrators (if any), which avoids potential overheads and allows users to install the entire platform on resource-constrained, low-power edge devices. For the implementation of the platform, we used Elixir [55], which is a functional, concurrent, high-level general-purpose programming language that runs on the BEAM virtual machine [107] (used by the Erlang language). Specifically, Elixir and the BEAM allowed us to simplify the creation and deployment of a distributed application without relying on container orchestration technologies, while retaining high performance, fault-tolerance, and resilience (provided by the BEAM's scheduler and lightweight processes, famous for being optimised for concurrent and distributed systems).

We represent in Figure 8.1 both the components that make up the platform's architecture and the typical flow developers and users follow to create and invoke functions. Architecture-wise, FunLess consists of mainly two components: the *Core* and the *Worker*, which we detail in the next parts of this section. Briefly, the *Core* acts

as an user-facing API to *i*) create, fetch, update, and delete functions and *ii*) schedule functions on workers. The *Worker* is the component deployed on every node tasked to run the functions; in the remainder, we refer to these nodes as *Worker*s. In addition to these two components, FunLess includes a *Postgres* database to store functions and metadata and *Prometheus* to collect metrics from the platform.[6]

FunLess is an open-source project and both its source code [37] and documentation [38] are publicly available.

Core. The *Core* is the controller of the platform. It exposes an HTTP REST API to the users, handles authentication and authorization, and manages functions' lifecycle and invocations.

Although the *Core* implements the main coordination logic and functionalities of FunLess, it is a lightweight component. For instance, on a Raspberry Pi 3B+ its local bare-metal deployment (that includes the database, the monitoring system and the underlying operating system and services) occupies ca. 600 MB of RAM when idle.

Functionality-wise, FunLess users create a new function by compiling its source code to Wasm—using either the language's default compiler (for Rust), an alternative one (for Go), or an external tool (for JavaScript)—and uploading the resulting binary to the *Core*, assigning to it a name. Users can group functions in modules and, when uploading a function, they can optionally specify which module the function belongs to. Moreover, users should also specify the amount of memory reserved for the execution of the function.

Looking at the steps reported in Figure 8.1, once the *Core* receives the request to create a function (1. Upload), it stores its binary in the database (2. Store). Fetch, update, and deletion happen via the assigned function name. When the *Core* successfully creates a function, it notifies the *Worker*s (3. Broadcast) to store a local copy of the function binary (4. Cache) compiled from the source code with the given metadata (i.e., module and function names). This push strategy helps to reduce part of the overhead of cold starts. Indeed, most FaaS platforms follow a pull policy where, if the execution nodes do not have the function in their cache (e.g., it is the first time they execute), they fetch, cache, and load the code of the function, undergoing latency. The small occupancy of Wasm binaries makes it affordable for FunLess to employ a

[6]Resp. found at `https://www.postgresql.org/` and `https://prometheus.io`.

push strategy, helping to reduce cold-start overheads.

Since both the *Core* and the *Worker*s run on the BEAM, these components communicate via the BEAM's built-in lightweight distributed inter-process messaging system, avoiding the need (complexity, weight) for additional dependencies for data formatting, transmission, and component connection.

When a function invocation reaches the *Core* (5. Invoke), the latter checks the existence of the function in the database and retrieves its code (6. Retrieve). If the function is present in the database, the *Core* uses the most recent metrics—we represent the pushing of the data, updated every 5s by default, from Prometheus to the Core with the dashed line in Figure 8.1—to select on which of the available *Worker*s to allocate the function (7. Request). The selection algorithm starts from the *Worker* with the largest amount of free memory to the one with the smaller. If no worker has enough memory to host the function, the invocation will return with an error.

After the *Worker* successfully ran the function (we detail this part of the workflow in the section about *Worker*s, below) it sends back to the *Core* the result (if any), which the *Core* relays back to the user (10a/13b. Reply). If no *Worker* is available at scheduling time or there are errors during the execution, the *Core* returns an error.

Another important feature of FunLess is that the *Core* can automatically discover the *Worker*s in its same network. This feature derives from Elixir's libcluster library[7], which provides a mechanism for automatically forming clusters of BEAM/Erlang nodes. Technically, when deployed on bare metal, FunLess follows the Multicast UDP Gossip algorithm of the library, to automatically find available workers. Instead, when deployed using Kubernetes, FunLess relies on the service discovery capabilities of the container orchestration engine to connect the *Core* with the *Worker*s, paired with the "Kubernetes" modality of the library. Users can manually connect *Worker*s from other networks via a simple message (e.g., a ping) thanks to the BEAM's built-in capability of connecting to other BEAM nodes.

Worker. The *Worker* executes the functions requested by the *Core*. The *Worker*s employs Wasmtime, a standalone runtime for Wasm and WASI by the Bytecode Alliance [21]. The main reasons behind this choice come from the ease of integration, amount of contributors, and security-oriented focus of the project. While *Worker*s

[7] `https://hexdocs.pm/libcluster/readme.html`.

integrate Wasmtime, we modelled their architecture to abstract away the peculiarities of specific Wasm runtimes so that future variants can use different runtimes and even extend support for multiple ones (possibly letting users specify which one to use). When a *Worker* receives a request from the *Core* to execute a function (7. Request), it first checks whether it has a cached version of the function's binary (8. Retrieve). If that is the case, it loads and runs the function's binary and returns to the *Core* the result of the computation (9a. Result). If the *Worker* does not find the code of the function in its local cache, it contacts the *Core* (9b. No Code Message), which responds with a request that carries the code of the function to the *Worker* (10b. Request with Code). Upon reception, the *Worker* compiles the code, caches the binary for future invocations (11b. Cache), loads it to run the function, and relays the result to the *Core* (12b. Result). The above mechanism is an important advantage afforded by FunLess for the edge case. Function fetching (if needed) transmits small pieces of binary code (rather than heavyweight containers). Wasm binaries achieve the two-fold objective of having *Worker*s run functions on different hardware architectures (e.g., AMD64, ARM) and allowing users to write their functions once, knowing that they will execute irrespective of the hardware of the *Worker*.

Summarising, fetching and precompiling (if any, depending on cache status) constitutes most of the "cold start" overhead in FunLess, which the platform greatly reduces w.r.t. alternatives relying on containers (which are heavier both in terms of bandwidth and memory occupancy).

Regarding caching and eviction, *Worker*s set a threshold for the cache memory (configurable at deployment time). If the storing of a new function exceeds that threshold, the *Worker* evicts the function with the longest period of inactivity (invocation- or update-wise). Additionally, *Worker*s automatically evict functions if inactive for a set amount of time (by default, 45 minutes).

Storage. FunLess relies on PostgreSQL as its primary storage solution. The storage component maintains the state of the platform through a well-structured database schema that reflects the hierarchical organization of functions and modules. The database schema centers around two main entities: the *module* and *function* tables. The *module* table serves as a logical container for grouping related functions, and storing essential metadata such as the module name and associated user owner. The *function*

table maintains records of all functions in the platform, storing both metadata and the actual WebAssembly binary. Each function entry includes the compiled WebAssembly binary stored as a binary large object (BLOB). This storage architecture supports the efficient operation of the *Core* component, enabling quick retrieval of function binaries during cold starts and providing robust persistence for the platform's state.

Metrics. The architecture integrates a Prometheus service as its metrics collection and monitoring system, enabling observability of the platform's performance and resource utilization. Prometheus implements a pull-based architecture where it periodically scrapes metrics from the *Worker* components through exposed HTTP endpoints. The *Worker*s report operational metrics including memory usage and CPU utilization. These metrics are collected at configurable intervals (default: 5 seconds) and stored in Prometheus's time-series database, and the *Core* executes PromQL queries to retrieve up-to-date performance metrics, which inform its scheduling decisions. The monitoring system also facilitates platform maintenance and troubleshooting by providing historical performance data and enabling the detection of potential bottlenecks or resource constraints.

8.3.1 Design choices and limitations

Since a small resource footprint and simplicity (of architecture and computation) are the driving principles behind FunLess' implementation, we favoured design choices (both w.r.t. the components in the architecture and the internal implementation) that introduced the least complexity while affording flexibility (of implementation and deployment). Below, we discuss the main aspects that FunLess trades off for the above benefits.

Language support. FunLess requires functions to be compiled to Wasm to execute them. Moreover, for the Wasm binary to properly integrate with the *Worker*, it needs to expose a specific function that acts as a "wrapper" for the user's function. The wrapper performs input and output (de)serialisation, and is not a standard feature of Wasm modules. Therefore, FunLess provides a wrapper for each supported language—depending on the language, a wrapper can be a library, macro or compiler extension. While offering support for different languages is not essential for this presentation,

FunLess already supports three languages: Rust, Go and JavaScript—and we planned support for more in the future. Specifically, we chose Rust for its performance, its growing developer community, and its ease of compiling to Wasm; similarly, Go is famous for its performance and widespread use in cloud computing; lastly, JavaScript is one of the most popular languages in software development.

Resilience. FunLess's *Core* component, which acts as the sole scheduler and holder of the platform's state, is not replicated. On the one hand, this reduces the footprint of the platform since users just need to deploy one *Core*. On the other hand, the *Core* is a single point of failure of the architecture. The BEAM opportunely guarantees fault-tolerance, so that the *Core* can recover from software crashes. However, the platform would stop working properly if the hardware hosting the *Core* failed. On software crashes, the only data lost are the invocations in transit (which the users would notice as timed out), but the rest of the system would recover (normal functionality, connections to the *Worker*s, metrics, and storage), following the connection protocols mentioned above.

Robustness. FunLess implements an at-most-once message relay policy, hence, lost messages between the *Core* and *Worker*s imply the failure of the invocation. Implementing more robust semantics, e.g., at least once, would require the inclusion of a message broker, increasing the load on nodes and the architecture's complexity.

Retry policies. The *Core* does not implement retry policies. Thus, if a function's execution fails on the chosen *Worker* or that *Worker* becomes unresponsive, the *Core* does not try to run the function on another worker. Implementing retry policies would increase the complexity platform-wide. Specifically, the *Core* would need to keep track of the state of function invocations, increasing the amount of coordination/messages with the *Worker*s. This extension would also increase the amount of data and interactions with the database (needed to enforce the transactional management of functions' state and stave off the risk of losing this data due to crashes) and further complicate the *Core*'s implementation to manage back-off strategies and execution time limits. Nonetheless, we plan to implement retries with an "opt-in" approach (the BEAM already provides some building blocks for the task, used to implement function timeouts and monitoring), giving users the flexibility to choose between a lighter setup or increased reliability.

8.4 Energy Consumption Comparison

We used FunLess to perform a comparative analysis of energy consumption in FaaS platforms. To assess the efficiency in energy usage in these kinds of platforms, we compared different implementations of the same use case: long-running containerized services and serverless computing functions. For the serverless approach, we further consider two different implementations: one using OpenWhisk (functions backed by containers) and the other with FunLess (functions in WebAssembly).

8.4.1 Use Case

The use case involes a simple distributed architecture for a laboratory environment. The architecture, depicted in Figure 8.2, consists of a data processing pipeline composed of three main services.

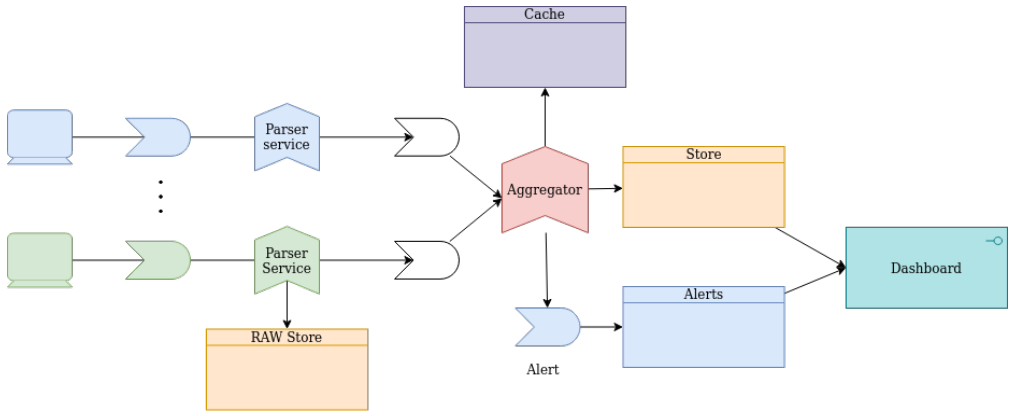


Figure 8.2: Architecture of the laboratory environment use case.

It starts from the edge devices, which include various sensors such as temperature, pressure, and humidity sensors. These sensors collect environmental data from the laboratory and send it to a Parser service. The Parser service acts as a data receiver, formats the data into a standardized JSON package and sends it to the Aggregator service. The Aggregator service collects the several JSON packages from the Parser services (one for each sensor) and prepares the data for further processing into a bundle. The data is sent to storage that a Dashboard service in the cloud can retrieve to provide a real-time visualization of the sensors for monitoring purposes. Additionally, the data can be stored for future analysis and retrieval.

The analysis focuses on the three core services, therefore they have been implemented using the three different approaches: as containerized services, as OpenWhisk functions, and as Wasm functions for FunLess. The implementation of the services simulate the workload for benchmarking purposes. In the case of the containerized services the implementation is in Elixir, chosen for its built-in distribution capabilities. The Parser service performs a creation of a JSON object simulating the reception of data from a sensor, and is configured to forward the data to the Aggregator service. The Aggregator service simulates the aggregation of the data from the Parser services, and performs a simple mathematical computation in a loop with 1 million iterations to introduce deliberate processing delays to mimic real-world data processing scenarios. Finally, the Dashboard service simulates the visualization of the aggregated data by generating an HTML page. For the serverless implementations, we use JavaScript for the function logic with follow an equivalent logic to the containerized services. For FunLess, the relative functions are compiled to WebAssembly. Following this configuration, a request to the Parser service triggers the usage of the Aggregator service as well, while the Dashboard service can remain idle until it is accessed by a user.

8.4.2 Evaluation

Test Setup To monitor the energy consumption we used PowerAPI [35]. PowerAPI is an open-source framework designed to monitor and analyze the energy used of software systems. PowerAPI utilizes the HardWare Performance Counter (HWPC) Sensor to track the power consumption of Intel CPUs and estimate the energy usage via two different models: RAPL (Running Average Power Limit) formula and the SmartWatts formula. The former is a feature provided by Intel processors that allows for the measurement of power from the HWPC sensor, while the latter is a software-defined power meter based on the PowerAPI toolkit, which includes HWPC metrics together with other system events. In our evaluation, we measured the consumed milliwatts (mW) of each service for a fixed period of time, sampling the power consumption every second. For the comparisons, we measured the energy consumption of the indivual services for each approach. For the containerized service architecture we measured the three core services, for OpenWhisk we measured the Controller, CouchDB, the Invoker, Kafka, and Zookeeper services, and for FunLess we measured the Core, Postgres,

Prometheus, and Worker services. Finally, we analysed the total energy consumption of the three approaches by summing the measured milliwatts of the individual services. The tests we conducted using Apache JMeter to send requests to the services.

Scenarios The tests were conducted on a single node equipped with an Intel i7 processor with 12 cores and 32 GB of RAM. The test runs included three scenarios: *i*) idling, where no requests were sent to the services; *ii*) constant workload, where requests were sent at a constant frequency (5000 requests with 5 req/s); *iii*) spiked workload, which involved bursts of requests potentially leading to cold starts for the FaaS platforms.

Results

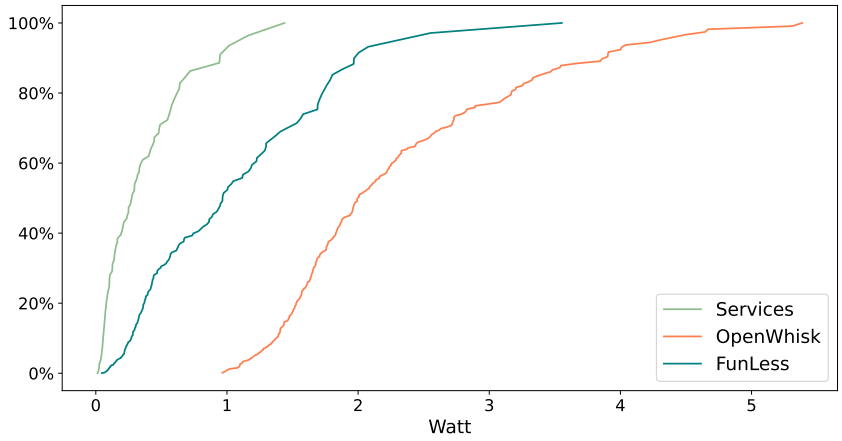


Figure 8.3: Energy-usage sample distribution (idle scenario).

Idle Scenario Starting with the idle scenario, we measured the energy used for 5 minutes, totalling 300 samples. Figure 8.3 shows the plot-line distribution of the energy consumption, and we report in Table 8.1 the aggregated average, standard deviation, and median. The figure shows the percentage of samples that fall below a certain value. FaaS platforms exhibit higher consumption due to the several long-running services that compose them, with OpenWhisk having a higher baseline consumption than the other two approaches (generally above 1000 mW), and the containerized services showing the lowest consumption.

	Average (mW)	**Std. Dev. (mW)**	**Median (mW)**
Cont. Services	134.40	185.18	67.2
OpenWhisk	2001.97	807.89	1757.3
FunLess	493.84	520.02	308.98

Table 8.1: Summed energy usage of the three approaches in the idle scenario.

Constant Workload Scenario In this scenario we sent 5000 requests to the services at a constant rate of 5 requests per second, obtaining a total of 1000 samples. As shown in Figure 8.4, the energy consumption difference between the three approaches are less pronounced. OpenWhisk still has the highest consumption, although in the best cases it is similar to FunLess, around 2000 mW. The metrics reported in Table 8.2 reiterate the trend, although FunLess shows more stability than OpenWhisk.

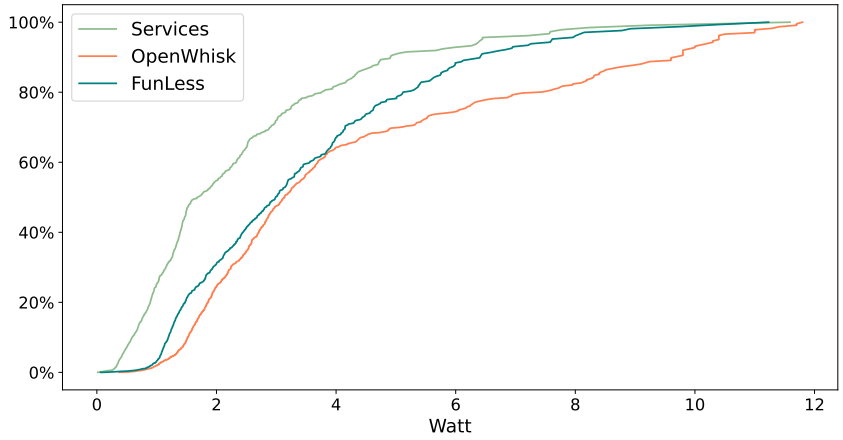


Figure 8.4: Energy-usage sample distribution (constant workload scenario).

	Average (mW)	**Std. Dev. (mW)**	**Median (mW)**
Cont. Services	1202.56	1235.59	858.35
OpenWhisk	2766.14	2065.93	2110.00
FunLess	2299.30	1630.66	1708.9

Table 8.2: Summed energy usage of the three approaches in the constant workload scenario.

Spike Calls Scenario The trend changes significantly in the spiked workload scenario. This test was run with 1 request per second for 30 seconds, followed by a sudden increase to 10 requests per second for the next 30 seconds. After that, the rate dropped back to 1 request per second for a minute, and finally, there was a spike to 100 requests per second for another minute. Figure 8.5 plots the energy usage over time, showing the two spikes

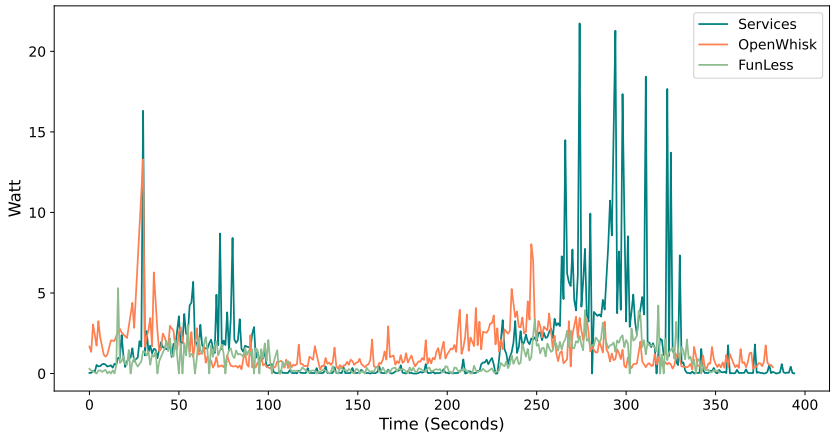


Figure 8.5: Energy-usage over time (spiked workload scenario).

as they occur. In this situation the FaaS platforms handled the change in workload more efficiently with FunLess keeping a lower energy consumption than OpenWhisk. OpenWhisk also managed the spikes relatively well, except for the first requests where it started scaling the function containers and cold-starts significantly impacted energy usage. On the other hand, the containerized services performed the worst, especially during the second, more intense spike. Figure 8.6 shows the energy-usage sample distribution for the spiked workload scenario. The summarized energy usage metrics are reported in Table 8.3.

	Average (mW)	**Std. Dev. (mW)**	**Median (mW)**
Cont. Services	1775.46	3054.79	681.25
OpenWhisk	1397.77	1278.01	965.00
FunLess	976.64	817.73	812.10

Table 8.3: Summed energy usage of the three approaches in the spiked workload scenario.

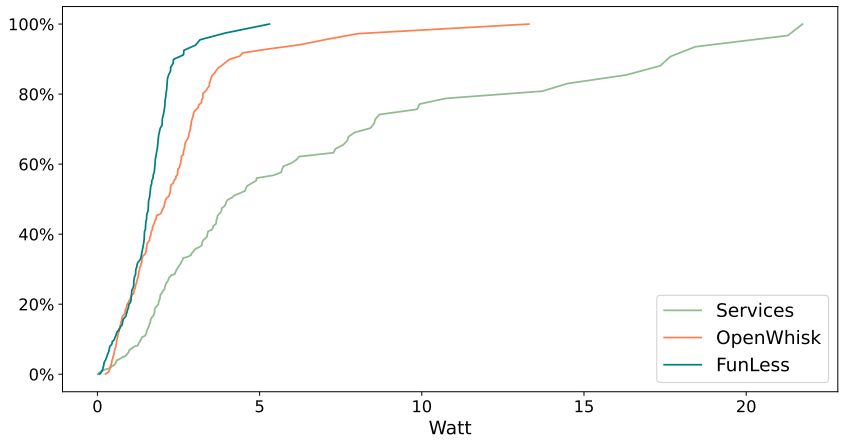


Figure 8.6: Energy-usage sample distribution (spiked workload scenario).

In this last test OpenWhisk showed a higher consumption when it started scaling due to the cold-starts. The experiments show that the OpenWhisk Invoker reaches a peak of 7.4 Watts when the cold-start triggering requests are sent. Moreover, each function container consumes an average of 200 mW in this use case. In FunLess' case, the Worker service reaches a measured peak of 4.5 Watts with cold-starts requests. Cold-starts for Wasm functions happen when the worker has to compile the Wasm module and instantiate the function in memory for the first request. Since the Wasm functions are executables handled by the Worker, there is no extra overhead for the cold-starts.

8.5 Conclusions

We presented FunLess, a FaaS platform tailored to respond to recent trends in serverless computing that advocate for extending FaaS to cover private edge cloud systems, including Internet-of-Things devices. The motivation behind the shift towards private edge cloud systems includes reduced latency, enhanced security, and improved resource usage. Unlike existing solutions that rely on containers and container orchestration technologies for function invocation, FunLess leverages Wasm as its function-execution runtime environment. The reason behind this choice is to reduce performance overheads that can prevent resource-constrained devices from running FaaS systems. Wasm's fundamental feature exploited by FunLess is its lightweight, sandboxed runtime, which allows the platform to run efficiently functions in isolation on constrained devices at the edge. Thus, Wasm provides a portable, homogeneous way for developers to implement and deploy their functions among clusters of heterogeneous devices (write once, run everywhere), simplifying platform deployments, offering flexibility in deployment options, and mitigating cold start issues. FunLess is also developed with support for APP and its extension in mind, which opens up the possibility of having a built-in and deeply integrated system for customizing scheduling behavior. It already provides support for APP and will be extended to support the entire family of APP languanges.

As future work, we plan to integrate new versions of Wasmtime and, with it, native support for HTTP and other optimisations and features of the new releases and support for the WASI runtime. Indeed, many current Wasm runtime implementations miss features like interface types, networking support in WASI multi-threading, atomics, and garbage collectors. Besides Wasmtime, other projects are developing new, optimised, and extended Wasm runtimes, which FunLess can leverage to increase its performance (and adapt it to different application contexts). For example, the support for garbage collection can lead to improved JavaScript runtimes and increase the performance of this kind of functions.

From the point of view of feature support, we deem supporting function composition in FunLess both important for the users and beneficial for performance. Indeed, FunLess currently supports function composition by publicly exposing the functions in a flow and chaining them via their public endpoints. In the future, we propose to study how technologies like FaaSFlow [67], Palette [2], AWS Step Functions [93], Azure Durable

Functions [20] work and integrate them into FunLess. In particular, since FunLess uses Wasm, an interesting direction is exploiting memory sharing to have Wasm functions in a flow to avoid the overhead of network communication by letting chained Wasm functions work on the same memory block to store and retrieve their data.

We also plan to improve the reliability of the platform, allowing the support of retry policies for failed invocations, at-least-once message delivery, and the replication of the *Core* components. Following the principles of simplicity and versatility that guided the development of FunLess, we propose to tackle these extensions as optional features to support flexible deployments, adaptable to the different application contexts (cloud, edge, on resource-constrained devices).

Finally, we plan to ease the deployment of FunLess by supporting other tools like, e.g., Nomad [78] and optimise the platform for edge devices by using, e.g., Nerves [77] to further minimise the overhead on bare-metal deployment.

Chapter 9

Discussion and Conclusion

In this chapter, we discuss the positioning of the work presented in this thesis with respect to the state of the art and related work, and conclude by drawing final takeaways and future research directions.

9.1 Related Work

The industrial adoption of Serverless is spreading [14] and it is a hot research topic due to its "untapped" potential [13, 46, 47, 54]. Cloud-edge deployments for FaaS platforms are gaining traction, as they ease the development of applications that need to span multiple locations and devices, while retaining the benefits of Serverless Computing (such as event-driven function invocations and automatic scaling). In this context, function scheduling becomes an important aspect to consider, and numerous research efforts have been dedicated to improving it.

Serverless Functions Optimizations One of the main approaches explored in the literature to improve Serverless performance through function scheduling comes from improving the warm- vs cold-start of functions [46, 54]. Those techniques mainly regard containers re-utilisation and function scheduling heuristics to avoid setting up new containers from scratch for every new invocation. However, other techniques have been proposed in the literature. Mohan et al. [74] present an approach focused on the pre-allocation of network resources (one of the main bottlenecks of cold starts) which

are dynamically associated with new containers. Abad et al. [1] present a package-aware scheduling algorithm that tries to assign functions that require the same package to the same worker. Suresh and Gandhi [109] present a function-level scheduler designed to minimise provider resource costs while meeting customer performance requirements. In this direction of improving scheduling by reducing cold starts, Shahrad et al. [97] introduce an empirically-informed resource management policy that mediates cold starts and resource allocation. Silva et al. [100] propose a solution based on process snapshots: when the user deploys the function, they generate/store a snapshot of the process that runs that function and, when the user invokes the function, they load/run the related snapshot.

Topology-awareness One work close to tAPP is by Sampé et al. [91], who present an approach that allocates functions to storage workers, favouring data locality. The main difference with our work is that the one by Sampé et al. focusses on topologies induced by data-locality issues, while we consider topologies to begin with, and we capture data locality as an application scenario. Banaei et al. [15] introduce a scheduling policy that governs the order of invocation processing, depending on the availability of the resources they use. Shillaker and Pietzuch [99] use state by supporting both global and local state access, aiming at performance improvements for data-intensive applications. Similarly, Jia and Witchel [52] associate each function invocation with a shared log among serverless functions. Additionally, approaches like Pheromone, by Yu et al. [127], combine local schedulers, which locally execute function workflows, and global coordinators, which offload the functions when local executors are busy. The local schedulers, combined with worker-specific shared-memory object stores, allow functions to rapidly exchange data without going through external storage.

Looking at other work that uses localities to improve FaaS performance, Lambdata [110] is an OpenWhisk extension that improves its performance considering locality and cold starts. Lambdata builds on top of both the OpenWhisk's Controller and Invoker components to allow users to annotate functions with explicit *data intents*, specifying which buckets they intend to use for reading and writing. Lambdata's approach is complementary to ours since APP (and its extensions) allows for more fine-grained control over function-Invoker assignment, while Lambdata infers such assignments from annotations. Besides resource re-utilisation, other approaches tackle the problem

of optimising function scheduling with new balancing algorithms. Steint [105] and Akkus et al. [4] proposed new algorithms for Serverless scheduling, respectively using a non-cooperative game-theoretic load balancing approach for response-time minimisation and a combination of application-level sandboxing with a hierarchical message bus.

Function workflows Morevoer, recent developments in FaaS involve the definition and management of function compositions or workflows, exemplified by AWS Step Functions [93] and Azure Durable Functions [9]. The fundamental concept beyond these advancements is to allow users to specify workflows by combining functions with branching logic, parallel execution, and error-handling capabilities. Then, the orchestrator or controller of the platform uses the defined workflow to oversee function executions, managing aspects such as retries, timeouts, and error resolution. We consider aAPP to be orthogonal to the function composition/workflows. Indeed, assuming a workflow is available, the orchestrator developed for handling serverless workflows should be extensible with an aAPP-like script to specify where to schedule the functions within a given workflow. Future work on this integration would support the enforcement of even more expressive policies than aAPP, like preventing function instances of the same workflow from sharing nodes. Steinbach et al. model function composition with TppFaaS [106], where they use Temporal Point Processes. They require no explicit locality requirements or configurations, and the user mostly relies on the accuracy of the underlying model. While the authors only tested their proposal in terms of accuracy over generated trace datasets—i.e., they did not apply it to locality issues—we see their approach interesting for applications for predictive scheduling and scaling. Kotni et al. [60] present an approach that schedules functions within a single workflow as threads within a single process of a container instance, reducing overhead by sharing state among them. Baldini et al. [14] demonstrate that Serverless function composition requires a careful evaluation of trade-offs, identifying three competing constraints that form the "Serverless trilemma", i.e., that without specific run-time support, compositions-as-functions must violate at least one of the three constraints. To solve the trilemma, they present a reactive core of OpenWhisk that enables the sequential composition of functions. Inspiring approaches in this direction are by Pubali et al. [29], who present a serverless platform where developers can constrain the information flow among functions to avoid attacks due to container reuse and data exfiltration, and by Dehury et al. [33],

who propose an extension of the TOSCA standard to control the flow of data inside Cloud applications with serverless components.

Multi-cloud and federated FaaS Beyond single-cloud deployments—i.e., which require coordination between different providers—we mention xAFCL [87] and SkyPilot [125] (although the latter is not directly related to FaaS). xAFCL [87] handles invocations over several FaaS providers, to optimise the execution of function workflows by estimating the duration of each function and forwarding its invocation to the appropriate provider. SkyPilot [125] follows a similar approach, but it acts as middleware between the user and several cloud providers, to dynamically select the appropriate target for requests according to cost, latency, and security requirements. Both xAFCL and SkyPilot work at a higher level of abstraction compared to our APP family of languages, intervening between the user and the target platform, and one could follow their approach to coordinate work between APP-based OpenWhisk and commercial solutions. Also an interesting domain of application is that of Sky Computing [108], where brokers handle the placement and oversee the execution of cloud jobs over multiple cloud providers. In the case of FaaS, we mention funcX [25], which is a federated serverless solutions that allows users to register their infrastructure as part of the platform's deployment and run their functions on any node they are authorised to access. While OpenWhisk is not suitable for such an approach (since it has no notion of federation), one can apply tAPP to this domain by employing topology-aware scheduling policies when users wish to run a function on a certain endpoint or group of endpoints. We also mention a work by Nardelli and Russo [76], which explores the concept of a decentralised serverless platform, where each node acts as entrypoint, and can either compute functions locally or offload them to other nodes. The scheduling in this case is completely automatic, and relies on data access probability estimates to predict the optimal node for function invocation. While tAPP is based on a more centralised architecture, it can be easily extended to target zones directly, without relying on a separate controller, to integrate user knowledge with the existing estimates.

Affinity-awareness Proposals in the direction of affinity-awareness in Serverless applications come from the neighbouring area of microservices—the state-of-the-art

style for cloud architectures. Baarzi and Kesidis [10] present a framework for the deployment of microservices that infers and assigns affinity and anti-affinity traits to microservices to orient the distribution of resources and microservices replicas on the available machines; Sampaio et al. [90], who introduce an adaptation mechanism for microservice deployment based on microservice affinities (e.g., the more messages microservices exchange the more affine they are) and resource usage; Sheoran et al. [98], who propose an approach that computes procedural affinity of communication among microservices to make placement decisions. Looking at the industry, Azure Service Fabric [70] provides a notion of *service affinity* that ensures that the replicas of a service are placed on the same nodes as those of another, affine service. Another example is Kubernetes, which has a notion of *node affinity* and *inter-pod (anti-)affinity* to express advanced scheduling logic for the optimal distribution of pods [64]. Overall, the mentioned work proves the usefulness of affinity-aware deployments at lower layers than FaaS (e.g., VMs, containers, microservices) and compels a discussion on the interplay between aAPP and IaaS/CaaS-level affinity. Another interesting proposal, Palette [2], uses optional opaque parameters in function invocations to inform the load balancer of Azure Functions on the affinity with previous invocations and the data they produced. While Palette does not support (anti-)affinity constraints, it allows users to express which invocations benefit from running on the same node. We deem an interesting future work extending aAPP to support a notion of (anti-)affinity that considers the history of scheduled functions.

Cost-awareness Regarding cost-awareness, to the best of our knowledge, ours is the first work that uses cost equations of functions to govern serverless scheduling. Some of the mentioned works focus on applying static analysis techniques for optimising serverless and cloud computing. For instance, Wang et al. [116] use static control and data flow analysis to enhance performance modelling of serverless functions, achieving accurate predictions. Obetz et al. [79] use service call graphs for static analysis of serverless applications, enabling various program analysis applications. Looking at the infrastructure underlying serverless, Garcia et al. [42] present a static analysis technique for computing upper bounds of virtual machine usage in cloud environments, using a technique similar to the one presented in Section 7.3. The inference of cost equations and their computation with cost analyzers has been also used for estimating

the computational time of programs in an actor model [65] and for analyzing updates of smart contracts balances due to transfers of digital assets [66]. Static-time techniques are also proposed in the field of Implicit Computational Complexity where type inference is used to derive (computational) costs of programs in a direct way, without resorting to cost analyzers. Similar to our approach, the techniques are applied to restricted languages where the cost analysis is decidable (e.g., *loop programs* as in [17]). It is worth to notice that, when such techniques are applied to `cAPP`, the resulting costs are less precise than those computed with cost analysers. One simple example is Listing 7.1, when computed according to [17], whose cost is $max(P,B)$ because, in loop programs, conditionals are always nondeterministic. Besides static analysis, other works used dynamic runtime analyses to visualise measure resource costs [115]. These tools operate by injecting instructions into a program or modifying its runtime to instrument real-time monitoring for collecting information about the behaviour of the program. Contrary to static analyses, dynamic ones requires modifying the runtime of the platform to collect the data needed by the analysis. Moreover, it requires the execution of the programs/functions over an exhaustive set of inputs, which makes the application of the technique more impractical (and could provide a partial "view" of the cases).

Cloud-Edge Serverless Platforms Looking at the work from the literature most closely related to FunLess, we have several proposals targeting edge and cloud scenarios. From the review by Cassel et al. [23], most of the solutions (86%) for IoT/edge rely on some container technology while promising technologies like WebAssembly and Unikernels represent only 2-3% of the proposals. Focusing on serverless platforms supporting Wasm runtimes, Hall and Ramachandran [45] are among the first to advocate WebAssembly as the enabling technology to avoid the overhead of containers, which substantially weigh on the limited hardware resources of edge computing environments. The authors presented a serverless platform that runs WebAssembly code within the V8 JavaScript engine for execution and sandboxing of functions. Differently from FunLess, they use a NodeJS runtime that embeds V8 for the running Wasm code. As the authors note [45], the nesting of these layers takes a conspicuous toll on the performance of the system. Gadepalli et al. [41] use WebAssembly to run and sandbox serverless functions. They target only single-host deployments, requiring the deployment of the entire platform on one node only. Moreover, they do not support WASI [123], thus

making their system potentially less portable. Gackstatter et al. [40] propose WOW, a WebAssembly-based runtime environment for serverless edge computing integrated within the Apache OpenWhisk platform. The authors introduce a new layer between OpenWhisk and different Wasm runtimes which enable the execution of Wasm functions. Compared to FunLess, WOW requires the deployment of a full installation (of a custom version) of the OpenWhisk platform which precludes the installation of the controller to low-power and memory-restricted edge devices.[1] Lucet [68] was used by Fastly to run Wasm on their commercial Compute platform. Lucet translated WebAssembly to native code, which was then executed using Lucet's runtime also on edge devices. Unfortunately, Lucet has reached end-of-life and is no longer maintained. Cloudflare Workers [28] is also a commercial serverless platform that supports the possibility of defining functions in Wasm and has native support for WASI since 2022.[2] Although the runtime part of this project has recently been made open-source,[3] the serverless platform is proprietary and closed-source. It is worth mentioning the work by Shillaker and Pietzuch [99] that, tangential to our proposal, concerns a Wasm-based serverless runtime that uses Wasm to achieve state sharing across functions—they allow the execution of functions that share memory regions in the same address space for possible performance benefits. On a similar note, Zhao et al. [131] present an OpenWhisk extension for confidential serverless computing that integrates a Wasm runtime. The authors propose a solution to construct reusable enclaves that enable rapid enclave reset and robust security to reduce cold start times. Although these kinds of proposals are orthogonal to FunLess, we see them as future optimisations that the usage of a Wasm function runtime can unlock for FunLess. Kjorveziroski and Filiposka [58] focus on serverless orchestration using Wasm and introduce a variant of Kubernetes that can orchestrate Wasm modules that are executed without containers. Interestingly, also Kjorveziroski and Filiposka report that Wasm tasks enjoy faster deployment times (two-fold) and at least one order of magnitude smaller artefact sizes, while still offering comparable execution performance. Finally, Tzenetopoulos et al. [112] analyse the performance of *Lean OpenWhisk*, an edge-focused variant of the Apache OpenWhisk

[1] We tried to deploy WOW on a multi-host cloud configuration for comparison purposes. Unfortunately, the deployment failed (the platform relies on an old and modified version of OpenWhisk that is not supported anymore, i.e., the last commit in the project is older than 2 years).

[2] `https://blog.cloudflare.com/announcing-wasi-on-workers`

[3] `https://blog.cloudflare.com/workerd-open-source-workers-runtime/`

serverless platform. Their variant of the platform coalesces the scheduling and execution components in a single entity, removes the message broker (Apache Kafka) from the deployment, and introduces changes to reduce OpenWhisk's overhead, making it better suited for resource-constrained devices.

9.2 Conclusions

The primary objective of this thesis was to bridge the gap between the abstraction provided by serverless platforms and the complexity of modern distributed systems to enable more efficient function scheduling and better resource usage. In order to achieve this goal, we proposed several works addressing *function-execution scheduling optimisation*. We first proposed a methodology that provides developers with a declarative language called APP to express scheduling policies for functions. We extended the scheduler of OpenWhisk to use APP-defined policies and empirically tested our extension on a use case that combines IoT, Edge, and Cloud Computing, contrasting our implementation with a naïve one using the vanilla OpenWhisk stack to achieve the same functional requirements. We then extended this language in several directions to explore different applicable constraints. We introduced tAPP, as a topology-aware APP where scripts can restrict the execution of functions within zones to help improve the performance (e.g., by exploiting data or code locality properties), security, and resilience of serverless applications. We again validated our approach by presenting a prototype tAPP-based OpenWhisk, which we used to demonstrate that tAPP allows for an easy deployment of cloud-edge serverless systems with typical topology-aware scheduling constraints that cannot be guaranteed by standard vanilla OpenWhisk deployments. We then presented an affinity-aware APP, aAPP, with an implementation and validation to effectively tackle the challenge of enforcing affinity and anti-affinity constraints in a FaaS platform. Our approach involved creating a aAPP-based OpenWhisk, which we used to demonstrate the effectiveness in reducing latency and tail latency in particular scenarios. These findings underscore the importance of considering affinity requirements, particularly in multi-zone execution contexts. As a last extension, we introduced a framework that lightens the burden on the shoulders of users by deriving cost information from the functions, via static analysis, into a cost-aware variant of APP that we call cAPP. We demonstrated that one can over-approximate, at

scheduling time, the overall latency endured by the invocation of a function `f` when running on a given worker and use this information to govern its scheduling. The extension adds new syntactic fragments to APP so that programmers can govern the scheduling of functions towards those execution nodes that minimise their calculated latency (e.g., increasing serverless function performance) and avoids running functions on nodes whose execution time would exceed a maximal response time defined by the user (e.g., enforcing quality-of-service constraints). The main technical insights behind the extension include the usage of inference rules to extract cost equations from the source code of the deployed functions and exploiting dedicated solvers to compute the cost of executing a function, given its code and input parameters. As a final contribution, we presented FunLess, a FaaS platform tailored to respond to recent trends in serverless computing that advocate for extending FaaS to cover private edge cloud systems, including Internet-of-Things devices. The motivation behind the shift towards private edge cloud systems includes reduced latency, enhanced security, and improved resource usage. Unlike existing solutions that rely on containers and container orchestration technologies for function invocation, FunLess leverages Wasm as its function-execution runtime environment. The reason behind this choice is to reduce performance overheads that can prevent resource-constrained devices from running FaaS systems. Wasm's fundamental feature exploited by FunLess is its lightweight, sandboxed runtime, which allows the platform to run efficiently functions in isolation on constrained devices at the edge. Thus, Wasm provides a portable, homogeneous way for developers to implement and deploy their functions among clusters of heterogeneous devices (write once, run everywhere), simplifying platform deployments, offering flexibility in deployment options, and mitigating cold start issues. The sum of these features makes FunLess a greener FaaS platform as evidenced by the results of our energy-consumption comparison. Finally, FunLess also features support for APP and `cAPP` as a first-class citizen, allowing developers to leverage the capabilities of customizable scheduling. By addressing these areas, this thesis contributes to the broader goal of making serverless computing more adaptive, efficient, and applicable to a wide range of use cases.

9.2.1 Future Work

We would like to investigate the separation of concerns between developers and providers, trying to minimise the information that providers have to share to allow developers to schedule functions efficiently, while, at the same time, hide the complexity of their dynamically changing infrastructure. We also plan to expand FunLess support to the other versions of the APP language, i.e. tAPP and aAPP, and to extend the range of tests both to include other aspects of locality (e.g., sessions) and specific components of the platform. Regarding tests, we remark on the general need for more platform-agnostic and realistic suites, to obtain fairer and thorough comparisons. We started to benchmark FunLess against existing serverless platforms and several deployment scenarios, considering private, public, and mixed cloud-edge configurations [86]. These preliminary experiments show that, particularly in edge scenarios, FunLess outperforms alternatives like OpenFaaS, Fission, and Knative in terms of memory footprint without substantial performance degradation. We also aim to extend our energy usage comparison work to a comprehensive study on energy consumption in FaaS platforms. This involves extending our benchmarks to include a wider range of platforms and configurations, and developing a more detailed understanding of the energy profiles of different workloads. By doing so, we aim to provide more accurate and actionable insights into the energy efficiency of serverless computing, which is increasingly important in the context of sustainable computing. Finally, we would like to support DevOps in the optimization of their serverless applications by studying and experimenting with heuristics and AI-based mechanisms that profile applications and suggest optimal policies. Similarly, scheduling policies could benefit from interactions with frameworks able to specify function compositions, e.g., Yussupov et al. [129] recently introduced a method for modelling and deploying serverless function orchestrations which one could use to extract execution dependencies among functions and inform the synthesis of policies that optimise the overall execution of compositions.

Bibliography

[1] Cristina L Abad, Edwin F Boza, and Erwin Van Eyk. Package-aware scheduling of faas functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 101–106, 2018.

[2] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose M. Faleiro, Gohar Irfan Chaudhry, Iñigo Goiri, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. Palette Load Balancing: Locality Hints for Serverless Functions. In *EuroSys*, pages 365–380. ACM, 2023.

[3] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI 2020*, 2020.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[5] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[6] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic

information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA):118:1–118:26, 2018.

[7] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.", 2010.

[8] Apache openwhisk. `https://openwhisk.apache.org/`.

[9] Microsoft Azure. Azure durable functions. `https://docs.microsoft.com/en-us/azure/azure-functions/durable/`.

[10] Ataollah Fatahi Baarzi and George Kesidis. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 427–441, 2021.

[11] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[12] Bojana Bajic, Ilija Cosic, Branko Katalinic, Slobodan Moraca, Milovan Lazarevic, and Aleksandar Rikalovic. Edge computing vs. cloud computing: Challenges and opportunities in industry 4.0. *Annals of DAAAM & Proceedings*, 30, 2019.

[13] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[14] 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*, pages 89–103, 2017.

[15] Ali Banaei and Mohsen Sharifi. Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform. *The Journal of Supercomputing*, 9 2021.

[16] 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. IEEE, 2019.

[17] Amir M. Ben-Amram and Lars Kristiansen. On the edge of decidability in complexity analysis of loop programs. *International Journal of Foundations of Computer Science*, 23(7):1451–1464, 2012.

[18] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

[19] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE cloud computing*, 1(3):81–84, 2014.

[20] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[21] Bytecode alliance. `https://bytecodealliance.org/`, 2024.

[22] Giuliano Casale, Matej Artač, W-J Van Den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, Mike Long, Vasilis Papanikolaou, Domenico Presenza, Alessandra Russo, et al. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020.

[23] Gustavo André Setti Cassel, Vinicius Facco Rodrigues, Rodrigo da Rosa Righi, Marta Rosecler Bez, Andressa Cruz Nepomuceno, and Cristiano André da Costa. Serverless computing for internet of things: A systematic literature review. *Future Gener. Comput. Syst.*, 128:299–316, 2022.

[24] Paul Castro, Vatche Isahagian, Vinod Muthusamy, and Aleksander Slominski. *Hybrid Serverless Computing: Opportunities and Challenges*, pages 43–77. Springer International Publishing, Cham, 2023.

[25] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 65–76, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Hybrid Cloud. The nist definition of cloud computing. *National institute of science and technology, special publication*, 800(2011):145, 2011.

[27] IBM Cloud. Ibm cloud functions. `https://cloud.ibm.com/functions/`.

[28] Cloudflare. How Workers works. `https://developers.cloudflare.com/workers/reference/how-workers-works/`, 1 2024.

[29] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*, pages 939–950, 2020.

[30] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. A declarative approach to topology-aware serverless function-execution scheduling. In *2022 IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 11–15, 2022*. IEEE, 2022.

[31] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. Allocation priority policies for serverless function-execution scheduling optimisation. In *Proc. of ICSOC*, volume 12571 of *LNCS*, pages 416–430. Springer, 2020.

[32] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[33] Chinmaya Kumar Dehury, Pelle Jakovits, Satish Narayana Srirama, Giorgos Giotis, and Gaurav Garg. Toscadata: Modeling data pipeline applications in TOSCA. *J. Syst. Softw.*, 186:111164, 2022.

[34] Nafise Eskandani and Guido Salvaneschi. The Wonderless Dataset for Serverless Computing. In *MSR*, pages 565–569. IEEE, 2021.

[35] Guillaume Fieni, Daniel Romero Acero, Pierre Rust, and Romain Rouvoy. PowerAPI: A Python framework for building software-defined power meters. *Journal of Open Source Software*, 9(98):6670, June 2024.

[36] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[37] FunLess Repository. `https://github.com/funlessdev/funless`, 2024.

[38] FunLess Website. `https://funless.dev/`, 2024.

[39] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. No more, no less: A formal model for serverless computing. In *Coordination Models and Languages: 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 21*, pages 148–157. Springer, 2019.

[40] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*, pages 140–149. IEEE, 2022.

[41] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In Dilma Da Silva and Rüdiger Kapitza, editors, *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, pages 265–279. ACM, 2020.

[42] Abel Garcia, Cosimo Laneve, and Michael Lienhardt. Static analysis of cloud elasticity. *Sci. Comput. Program.*, 147:27–53, 2017.

[43] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. Nanolambda: Implementing functions as a service at all resource scales for the internet of things. In *5th IEEE/ACM Symposium on Edge Computing, SEC 2020, San Jose, CA, USA, November 12-14, 2020*, pages 220–231. IEEE, 2020.

[44] Google cloud functions. `https://cloud.google.com/functions/`.

[45] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 225–236, 2019.

[46] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back, 2019.

[47] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, page 33–39, USA, 2016. USENIX Association.

[48] Kaixing Hong, Hai Huang, Jianping Zhou, Yimin Shen, and Yujie Li. A method of real-time fault diagnosis for power transformers based on vibration analysis. *Measurement Science and Technology*, 26(11):115011, oct 2015.

[49] Kaixing Hong, Ming Jin, and Hai Huang. Transformer winding fault diagnosis using vibration image and deep learning. *IEEE Transactions on Power Delivery*, 36(2):676–685, 2021.

[50] Michaela Iorga, Larry Feldman, Robert Barton, Michael J Martin, Nedim S Goren, and Charif Mahmoudi. Fog computing conceptual model. *NIST Special Publication*, 2018.

[51] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.

[52] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proc. of ACM SIGOPS SOSP*, page 691–707, New York, NY, USA, 2021. ACM.

[53] Apache jmeter. `https://jmeter.apache.org/`.

[54] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[55] Sāsa Jurič. *Elixir in action*. Manning, 2024.

[56] Daniel Kelly, Frank Glavin, and Enda Barrett. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 304–312. IEEE, 2020.

[57] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly as an enabler for next generation serverless computing. *J. Grid Comput.*, 21(3):34, 2023.

[58] Vojdan Kjorveziroski and Sonja Filiposka. Webassembly orchestration in the context of serverless computing. *J. Netw. Syst. Manag.*, 31(3):62, 2023.

[59] Knative. `https://knative.dev/`, 10 2024.

[60] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *Proc. of USENIX ATC*, pages 805–820. USENIX Association, 2021.

[61] Koyeb. `https://www.koyeb.com/`.

[62] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proc. of NetDB*, volume 11, pages 1–7, 2011.

[63] Kubernetes. `https://kubernetes.io/`.

[64] Kubernetes. Node Affinity. https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/, 2024.

[65] Cosimo Laneve, Michael Lienhardt, Ka I Pun, and Guillermo Román-Díez. Time analysis of actor programs. *J. Log. Algebraic Methods Program.*, 105:1–27, 2019.

[66] Cosimo Laneve and Claudio Sacerdoti Coen. Analysis of smart contracts balances. *Blockchain: Research and Applications*, 2(3):100020 (1–22), 2021.

[67] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 782–796. ACM, 2022.

[68] Lucet. https://github.com/bytecodealliance/lucet, 2020.

[69] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[70] Microsoft. Azure service fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview, 2024.

[71] Microsoft. Service affinity in Service Fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-advanced-placement-rules-affinity, 2024.

[72] Microsoft azure functions. https://azure.microsoft.com/.

[73] Chetankumar Mistry, Bogdan Stelea, Vijay Kumar, and Thomas F. J.-M. Pasquier. Demonstrating the practicality of unikernels to build a serverless platform at the edge. In *12th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2020, Bangkok, Thailand, December 14-17, 2020*, pages 25–32. IEEE, 2020.

[74] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile Cold Starts for Scalable Serverless. In *HotCloud*. USENIX Association, 2019.

[75] Mqtt.org, mq telemetry transport. `http://mqtt.org/`.

[76] Matteo Nardelli and Gabriele Russo Russo. Function offloading and data migration for stateful serverless edge computing. In Simonetta Balsamo, William J. Knottenbelt, Cristina L. Abad, and Weiyi Shang, editors, *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024, London, United Kingdom, May 7-11, 2024*, pages 247–257. ACM, 2024.

[77] Nerves project. `https://nerves-project.org/`, 2024.

[78] Nomad. `https://www.nomadproject.io/`, 2024.

[79] Matthew Obetz, Stacy Patterson, and Ana L. Milanova. Static call graph construction in AWS lambda serverless applications. In Christina Delimitrou and Dan R. K. Ports, editors, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019.

[80] Openfaas. `https://www.openfaas.com/`.

[81] Openstack. `https://www.openstack.org/`.

[82] OpenStack. Documentation. `https://docs.openstack.org/project-deploy-guide/openstack-ansible/ocata/app-advanced-config-affinity.html`, 2024.

[83] Repository with scripts to deploy tapp openwhisk. `https://github.com/mattrent/openwhisk-deploy-kube`, 04 2024.

[84] Giuseppe De Palma. https://github.com/giusdp/openwhisk, 2024.

[85] Giuseppe De Palma, Saverio Giallorenzo, Cosimo Laneve, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. An OpenWhisk Extension for Topology-aware Allocation Priority Policies. In *To appear in COORDINATION. Preprint availabe at* `https://arxiv.org/abs/2205.10176`, 2024.

[86] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. Funless: Functions-as-a-service for private edge cloud systems. *CoRR*, abs/2405.21009, 2024.

[87] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. xafcl: Run scalable function choreographies across multiple faas systems. *IEEE Transactions on Services Computing*, 16(1):711–723, 2023.

[88] The Coders Rocket. The dangers of serverless hosting: A cautionary tale for getting a 96,000 bill. `https://medium.com/@The-coders-rocket/the-dangers-of-serverless-hosting-a-cautionary-tale-for-getting-a-96-000-bill-2a` 2024.

[89] Mohammad Salehe, Zhiming Hu, Seyed Hossein Mortazavi, Iqbal Mohomed, and Tim Capes. Videopipe: Building video stream processing pipelines at the edge. In Dejan S. Milojicic and Vinod Muthusamy, editors, *Proceedings of the 20th International Middleware Conference Industrial Track, Davis, CA, USA, December 9-13, 2019*, pages 43–49. ACM, 2019.

[90] Adalberto R Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S Rosa. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, 10(1):1–30, 2019.

[91] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*, pages 121–133, 2017.

[92] Amazon Web Services. Aws lambda. `https://aws.amazon.com/lambda/`.

[93] Amazon Web Services. Aws step functions. `https://aws.amazon.com/step-functions/`.

[94] Amazon Web Services. How aws's firecracker virtual machines work. https://www.amazon.science/blog/how-awss-firecracker-virtual-machines-work, 2020. Accessed: 2024-10-05.

[95] Amazon Web Services. Anti-patterns in lambda-based applications. `https://docs.aws.amazon.com/lambda/latest/operatorguide/anti-patterns.html`, 2024.

[96] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. Serverless computing: A survey of opportunities, challenges, and applications. *ACM Comput. Surv.*, 54(11s):239:1–239:32, 2022.

[97] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. of USENIX ATC*, pages 205–218, 2020.

[98] Amit Sheoran, Sonia Fahmy, Puneet Sharma, and Navin Modi. Invenio: Communication affinity computation for low-latency microservices. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 88–101, 2021.

[99] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. of USENIX ATC*, pages 419–433. USENIX Association, 2020.

[100] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proc. of Middleware*, page 1–13, New York, NY, USA, 2020. ACM.

[101] 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 *ICFEC*, pages 17–25. IEEE, 2022.

[102] Khondokar Solaiman and Muhammad Abdullah Adnan. WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing. In *IC2E*, pages 144–153. IEEE, 2020.

[103] Benedikt Spies and Markus Mock. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.

[104] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[105] Manuel Stein. The serverless scheduling problem and noah. *arXiv preprint arXiv:1809.06100*, 2018.

[106] Markus Steinbach, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. Tppfaas: Modeling serverless functions invocations via temporal point processes. *IEEE Access*, 10:9059–9084, 2022.

[107] Erik Stenman. Beam: a virtual machine for handling millions of messages per second (invited talk). In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2018, page 4, New York, NY, USA, 2018. Association for Computing Machinery.

[108] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In Sebastian Angel, Baris Kasikci, and Eddie Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, pages 26–32. ACM, 2021.

[109] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th international workshop on serverless computing*, pages 19–24, 2019.

[110] 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.

[111] Not saying you should but we're told it's possible to land serverless app a '40k/month bill using a 1,000-node botnet'. `https://www.theregister.com/2021/04/21/denial_of_wallet/`, 2021.

[112] 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 *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt Am Main, Germany, June 24 – July 2, 2021, Revised Selected Papers*, pages 428–438, Berlin, Heidelberg, 2021. Springer-Verlag.

[113] Parichehr Vahidinia, Bahar J. Farahani, and Fereidoon Shams Aliee. Cold start in serverless computing: Current trends and mitigation strategies. In *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*, pages 1–7. IEEE, 2020.

[114] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[115] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 133–146, 2018.

[116] Runan Wang, Giuliano Casale, and Antonio Filieri. Enhancing performance modeling of serverless functions via static analysis. In Javier Troya, Brahim Medjahed, Mario Piattini, Lina Yao, Pablo Fernández, and Antonio Ruiz-Cortés, editors, *Service-Oriented Computing - 20th International Conference, ICSOC 2022, Seville, Spain, November 29 - December 2, 2022, Proceedings*, volume 13740 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 2022.

[117] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. *IEEE/ACM Transactions On Networking*, 24(1):190–203, 2014.

[118] Zicheng Wang. Can "micro vm" become the next generation computing platform?: Performance comparison between light weight virtual machine, container, and traditional virtual machine. In *2021 IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*, pages 29–34. IEEE, 2021.

[119] Wasmedge. https://wasmedge.org/, 8 2023.

[120] Wasmer. https://wasmer.io/, 8 2023.

[121] Wasmtime. https://wasmtime.dev/, 8 2023.

[122] Webassembly. https://webassembly.org/, 8 2023.

[123] Webassembly system interface. https://wasi.dev/, 8 2023.

[124] Qiaomin Xie, Mayank Pundir, Yi Lu, Cristina L Abad, and Roy H Campbell. Pandas: robust locality-aware scheduling with stochastic delay optimality. *IEEE/ACM Transactions on Networking*, 25(2):662–675, 2016.

[125] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[126] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All one needs to know about fog computing and related edge computing paradigms. *Journal of Systems Architecture*, 2019.

[127] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[128] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[129] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, and Frank Leymann. Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA. *Software: Practice and Experience*, 01 2022.

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

[131] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4015–4032. USENIX Association, 2023.