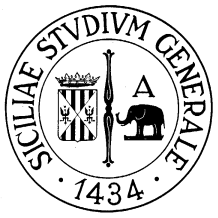

UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Ingegneria Elettrica, Elettronica ed Informatica

Dottorato di Ricerca in Ingegneria dei Sistemi, Energetica, Informatica e delle Telecomunicazioni
XXXIV Ciclo

Communication-aware management of SLAs for Cloud-Native Applications

On the road to AIOps. Smart orchestration strategies in Cloud and Edge computing.

Ing. Alessandro Di Stefano

Coordinatore
Chiar.mo Prof. Paolo Arena

Tutor
Chiar.mo Prof. Antonella Di Stefano

Abstract

Today, IT Operations teams have to face up with managing massive amounts of data generated by advanced distributed systems, workloads difficult to predict in time, security threats. They need to handle more incidents than ever before with strict service-level agreements (SLAs). Most of the current state-of-the-art techniques to handle SLAs for Cloud-Native applications are based mostly on severe human efforts.

Downtime can get expensive: companies can lose millions dollars per outage with a longer mean time to recovery due to the complexity of human debugging on complex distributed systems. In the landscape of hybrid clouds, multi-tenant environments, and Edge computing architecture, organizations need multiple strategies to get the desired quality of service. Capacity planning, resource utilization, storage management, anomaly detection, threat detection are just a few aspects that engineering teams should take into account to guarantee SLAs and sites reliability.

AIOps can empower software and service engineers to effectively build and operate cloud-native applications at scale with artificial intelligence (AI) and machine learning (ML) techniques.

This thesis explores the multitude of issues that constitute the landscape of cloud-native applications in Cloud and Edge computing scenarios.

The focus will be twofold: proposing containers allocation strategies where communication quality is considered first-citizen parameter, and opening a window in the novel field of AIOps by providing models, case studies, and strategies to perform smart orchestration of run-time workloads in PaaS clouds.

Contents

List of Figures

List of Tables

Part I

Preliminaries

Chapter 1

Introduction

Cloud Computing is the set of heterogeneous technologies that essentially provide hardware resources on-demand to run software by employing a pay-as-you-go business model.

Three main categories of clouds can be initially identified in the market:

- Infrastructure as a Service (IaaS): providers rent hardware resources (storage, virtualization, networking) to be managed by the customers IT teams;

- Platform as a Service (PaaS): services hosted by the Cloud Providers that enable a more fine-grained sharing of resources between customers asking for the execution of their workloads; users of this kind of clouds is in charge of developing software on the stack provided by the provider.

- Software as a Service (SaaS): services that directly consist of business-oriented software that is rent in form of license agreements for cus-

tomers that do not interact directly with the development of these services;

Another taxonomy distinguishes execution environments from the point of view of the customers in: (i) public (shared among different customers), (ii) private (the cloud environment itself is hosted by hardware in the behalf of the user company business itself), (iii) multi-cloud (involving one or more public clouds that interact each other), and (iv) hybrid cloud for which the user company of the cloud environment is spread across one (or more) public cloud(s) and one (or more) private data-centers.

Resource management in the Cloud must match the expectations of at least two stakeholders: the Cloud Provider and the Customer.

Customers are users of the Cloud and have the objective to execute their workloads with a given efficiency that allows them to provide services for their end-users: faults, outages, and slow-downs in a cloud does not let the companies to provide good Quality of Service or Experience for their end-users.

On the other hand, inadequate management of the customer's workloads can lead to waste of resources for the cloud providers, wrong exploitation of resources and non-optimal cost-benefit ratio.

1.1 Service-level Agreements

Today, the interactions between the Cloud service providers and the customers are based on service-level agreements (SLAs).

An SLA consists of business-related predicates for key performance indicators related to the technical requirements designed by the product engineers: the service-level objectives (SLOs).

To transform these technical parameters driving applications into SLOs is a big challenge that spawns over the whole life-cycle of the applications [14, 15].

The shift to infrastructure-as-a-code (IaaC) is one of the methodologies that enabled optimization of guarantees for SLAs. It required drastic changes on data-centers architectures: the main research contributions that enabled IaaC (and similar *aaC patterns) can be traced to Software-Defined Networking [16, 17] (SDN), Network Function Virtualization [18–20] (NFV), evolution of intra-datacenter networking with spine and leaf architectures [21], and finally, the implementation of orchestrators as Kubernetes and support decision systems to automatically handle operations on applications' life-cycle [15, 22], given the related SLOs [23].

New questions about what level of knowledge has to be shared between customers and Cloud providers have also arisen in recent years together with the study of adaptive schedulers and admission control systems [24].

1.2 Microservices and future internet issues

There is a new breed of technologies that are becoming mainstream in current infrastructures. Fog/Edge computing aims to partially move services from core cloud data centers into the Edge of the network [25, 26]. Consequently, network operators become active stakeholders in delivering not only communication services but also application services to the end-users.

Stake-holders that can be traced in the path of service delivery for end-users can be identified as (i) network providers (NP), (ii) resource providers (RP), and (iii) application providers (AP); collaboration be-

tween application providers and resource providers has to be taken into account together with collaboration between APs and NPs.

In this field, recent trends in software such as microservices foster the utilization of smaller deployment units and enable challenges for more innovative management of applications life-cycle. However, the increasing of the number of components also increased operations complexity.

The complexity of the resulting IT environment and services makes service orchestration a central task to coordinate and schedule operations on a myriad of distributed service components. Orchestration becomes even more challenging when different technologies are involved, requiring hybrid solutions that coordinate service provisioning and management, taking into account each technology's requirements and particularities.

1.3 From DevOps to AIOps

DevOps methodologies are based on automation, integration, monitoring and collaborations by exploiting continuous integration and deployment [27–29].

They are essential to guarantee release consistency and reduce the time to market deadlines.

DevOps tends to improve deployment and quality of software releases being evaluated through key performance indicators (KPIs) of the software life-cycle such as (i) deployment frequency, (ii) deployment time, (iii) change failure rate, (iv) time to detection, (v) availability, (vi) SLA compliance indices.

Those measures and the DevOps methodologies themselves are essential for the delivery and integration phases of the applications' life-cycle.

Beyond implementing new features that lead to the deployment of changes, run-time issues have to be assessed. Recent trends in the industry brought the academic research environment to explore autonomic

computing and automated orchestration of services to optimize the job of IT teams and minimize downtimes of applications.

In 2019, Gartner [30] came out with the AIOps keyword to invite researchers to combine DevOps methodologies with Big Data and Artificial Intelligence techniques to assess these issues.

DevOps speeds development by giving development teams more power to provision and reconfigure infrastructure, but IT still has to manage that infrastructure.

There is no widely accepted definition for AIOps yet. In [31], the authors define AIOps as techniques to empower software and service engineers to efficiently build, deploy and maintain services by using artificial intelligence algorithms.

New challenges for AIOps research spread from Security and Anomaly detection to new ways for optimal scheduling of tasks to people in IT Teams to solve detected problems in IT environments.

AIOps employs the use of big data platforms to aggregate siloed IT operations data in one place. This data can include (i) historical performance and event data, (ii) streaming real-time operations events, (iii) system logs and metrics, (iv) network data, and (v) incident-related data for ticketing and reporting.

The objectives lay in:

- Collect and aggregate the huge and ever-increasing volumes of data generated by multiple IT infrastructure components, applications, and performance-monitoring tools;

- Intelligent processing of “signals” out of the “noise” to identify significant events and patterns related to system performance and availability issues;

- Diagnose root causes and report them to IT for rapid response and remediation—or, in some cases, automatically resolve these issues without human intervention;
- Implementation of machine learning algorithms that can identify patterns, learning from past remediation measures, e.g., previous scripts executed, and automatically remedy the problem, thus reducing the need for manual intervention;
- Finally, modeling of SLOs as a code and automatic control of application life-cycle, e.g., auto-scaling, to guarantee SLAs;

By replacing multiple manual IT operations tools with a single, intelligent, and automated IT operations platform, AIOps enables IT operations teams to respond proactively to outages with a lot less effort.

It bridges the gap between an increasingly diverse, dynamic, and difficult-to-monitor IT landscape, on the one hand, and user expectations for little or no interruption in application performance and availability, on the other.

Some of the benefits AIOps methodologies are in the following, concerning two real scenarios that are leveraging this early technology:

- faster mean time to repair (MTTR): By cutting noise and correlating operations data from multiple IT environments, AIOps can identify root causes and propose solutions faster and more accurately than humanly possible. For example, Nextel was able to use AIOps to reduce incident response times from 30 minutes to less than 5 minutes by monitor 25k network devices and exploiting predictive operations management [32];
- move from reactive to proactive and predictive management: AIOps can provide predictive alerts that let IT teams address potential

problems before they lead to security issues or outages. After the COVID-19 outbreak, while Siemens workers started working remotely, they empowered their cyber-security solutions by gathering data from hardware devices such as laptops and PCs and analyzing those data to reveal potential threats.

1.4 Contributions and organization

The main contributions of this thesis led to two phases of the life-cycle of micro-services applications in Cloud environments: (i) initial, offline, placement of containers in a cluster with the emphasis of optimizing networking parameters in the scenarios of PaaS clouds and Edge Computing, and (ii) management, at run time, of the applications in generic hosting environments through graph-based monitoring, prediction of events (issues and anomalies), and autonomic strategies to take actions for keeping service-level indicators (SLIs) in the SLOs ranges defined by the applications' service-level agreements.

The thesis is organized into four parts. The purpose of part I is to describe the boundaries of the research area that has driven this Ph.D. course through a technological and academic research context at chapter 2. Part II will explore the work done in the above-cited field of the offline-placement and cloud-edge offloading strategies: chapter 4 describes a cloud-native scheduling approach for pods in PaaS based on Kubernetes when customer and resource providers share their knowledge to the aim of optimizing their QoS; chapter 5 presents MORA, a strategy for Cloud-Edge offloading based on the concept of *service elasticity* for container-based applications.

Part III will focus specifically on the field of AIOps for PaaS scenarios: it is being proposed an architecture, Ananke, in chapter 7, to enable modeling and processing of applications performance metrics through

multi-layer graphs. In the other chapters, *Ananke* represents the base architecture and model for case studies and strategies aiming to a clever orchestration of cloud-native applications. Chatper 8 proposes a framework for data analysis of monitoring data through Prometheus in *Ananke*. It also exposes the case study of an events classification strategy based on the Facebook Prophet [33] machine learning model, leading to an horizontal pod auto-scaler for Kubernetes. Chapter 9 continues proposing an improved strategy based on reinforcement-learning to not only scale pods deployed in a Kubernetes cluster but also execute actions to adapt the network configuration of the SDN connecting pods and physical hosts.

The last part of the thesis draws the line for research trends and future work in this field with a focus on the AIOps methodologies that the IT world expects to deploy in 30% of infrastructure by 2023 [30].

1.5 Acknowledgements

This work results from a training course in which you choose to place yourself personally at stake. It is the expression of a fruitful human and scientific experience from meeting many people and unique places. Many dear people have been close to me on this journey. Each of them devoted precious time to me, discussing and finding answers to my questions and ideas, sharing their ones. I sincerely thank my tutor, Prof. Antonella Di Stefano, for allowing me to get in touch with the world of research. Her tenacity, accompanied by his usual positivity, supported me even in the most challenging moments. Also, I express my gratitude for giving me part of his experience in cloud computing and distributed systems. Thanks especially for spending part of her time reading and discussing the drafts of the thesis work and other projects carried out during the Ph.D.

Special acknowledgement is for Giovanni Morana, that during these years has not been a simple guide and collaborator but also a friend, always available and ready to give me advice. An affectionate thanks go to Daniele Zito and Giovanni Cammarata who followed my path during my bachelor's degree. Thanks to Andrea Araldo from the Télécom Sud-Paris, Ben Steer and friends of the Queen Mary University of London, for turning on me more in the research world. I would never have been able to finish this job if I did not get the support of my girlfriend, Luna Raimondo. She followed me with love and patience in every moment of this thesis, celebrating the non-birthdays and talking to me about Cloud, Server, anything but her interest. Encouraging me even in the hardest moments. Her warmth and her subtle irony have facilitated my path. During the COVID-19 lockdown, the constant presence of Billy, a stray dog, and his grit and affectionate friendship consolidated from food and salami has always been valid support.

Chapter 2

Research Context

« What, if some day or night a demon were to steal after you into your loneliest loneliness and say to you: "This life as you now live it and have lived it, you will have to live once more and innumerable times more; and there will be nothing new in it, but every pain and every joy and every thought and sigh and everything unutterably small or great in your life will have to return to you, all in the same succession and sequence" »

Friedrich Nietzsche

2.1 Computing architectures

Today, IT Operations teams have to face up with the management of massive amounts of data generated by advanced distributed systems. They need to handle more incidents than ever before with strict service-level agreements (SLAs). Most of the current state-of-the-art techniques to handle SLAs for Cloud-Native applications are based on naive algorithms (e.g., to schedule containers in Kubernetes cluster) or human efforts. Moreover, many companies use ten or more tools for IT performance monitoring, and downtime can get expensive when companies lose million dollars per outage with a longer mean time to recovery due to the

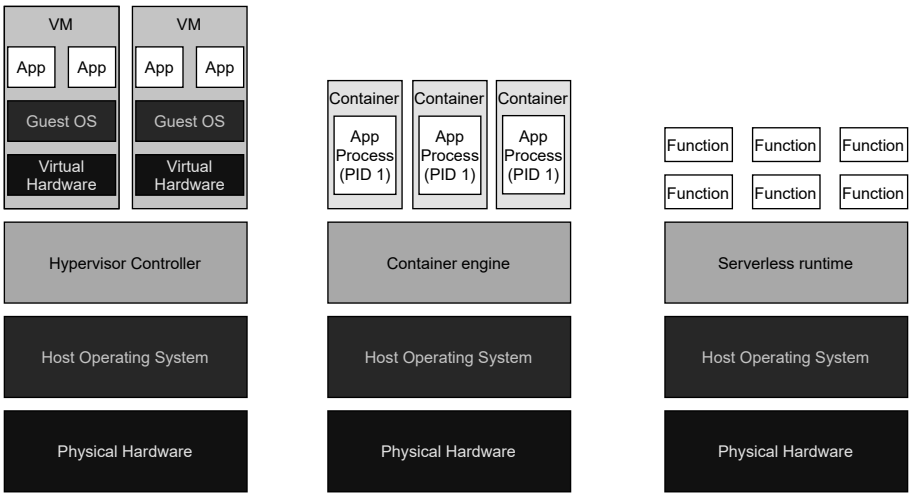

Figure 2.1: Bare metal to Serverless architecture road

complexity of human debugging on heavy systems analytics reports. On hybrid multi-cloud, multi-tenant environment organizations need even more strategies to get the desired quality of service, and this is important especially now that Edge/Fog computing architecture is arising in the 5G landscape. Capacity planning, resource utilization, storage management, anomaly detection, threat detection are just a few requirements that drive engineering teams to guarantee SLAs and sites reliability.

AIOps gives a prominent chance to empower software and service engineers to effectively build and operate cloud-native applications with artificial intelligence (AI) and machine learning (ML) techniques.

This thesis explores the multiple paradigms that constitute the landscape of cloud-native applications in Cloud and Edge computing scenarios. The focus will be twofold: solving containers allocation problems with communication quality being considered first-citizen and opening a window in the novel field of AIOps by providing models, case studies,

and strategies to perform clever orchestration of run-time workloads in PaaS clouds.

The concept of virtualization has its origins in the mainframe days in the late 1960s and early 1970s when early computer engineering industries invested in developing robust time-sharing solutions.

Time-sharing aims to share usage of computing resources among a large group of users. The objective is to increase the efficiency of both the users and the expensive computer resources they share. This model represented a breakthrough in computer technology: it became possible to use a computer without owning one.

Similar reasons drove virtualization for industry-standard computing in the late 1990s: the capacity in a single server is so large that it is almost impossible for most workloads to use it effectively. The best way to improve resource utilization, and at the same time, simplify data center management is through virtualization.

Data centers today use virtualization techniques to make abstraction of the physical hardware, create large aggregated pools of logical resources consisting of CPUs, memory, disks, file storage, applications, networking, and offer those resources to users or customers in the form of agile, scalable, consolidated virtual machines. Even though the technology and use cases have evolved, the core meaning of virtualization remains the same: to enable a computing environment to run multiple independent systems simultaneously.

The essential component in the virtualization stack is called *hypervisor*, or Virtual Machine Monitor (VMM).Type 1 hypervisors, in particular, create a virtual platform on the host server, on top of which multiple guest operating systems run, sharing the same resources.

During the last 30 years, several hypervisors have been built to handle the growth of the servers' hardware resources in the data centers through

the exploitation of multi-tenant architectures: VMware ESXi [34] and XEN Hypervisor [35] are just a few type 1 VMMs.

Virtualization allowed the explosion of the Cloud business model enabling large companies to own resources in over-provisioning and rent them together with intelligent solutions to guarantee multiple levels of quality of service, as stated in service-level agreements (SLA) built better to handle the gap between business agreements and technical orchestration of resources within one or more data centers.

2.1.1 Containerization

The history of containers technology is old; it can be traced back to the Unix *chroot* command and, later, to the FreeBSD jail(s). When Linux emerged as the dominant, free operating system, the technology enabling containers became widely used thanks to its underlying facilities: namespaces is a feature that allows partitioning of virtual kernel resources such as PIDs tables, user IDs, network names, and inter-process communication (IPC); *cgroups* is a more recent feature providing resource limiting, prioritization, accounting and, advanced control of checkpointing, freezing, and restarting processes (beneficial for VMs snapshotting and migration).

cgroups and *namespaces* are the building blocks for LXC, the operating system-level virtualization method to run multiple isolated Linux systems on a host by only using one Kernel and with no hierarchical hardware-level abstraction as in VMMs.

Many containers can be stored on a physical host with bulkier VMs.

The only limitation is that we must use the host's kernel to access the existing hardware components.

Docker is today an advanced technology that is widespread to reduce gaps between the development and the production environments so that

some deployment issues related to this gap can be avoided during the life-cycle of the application./

The advantages of a container-based cloud environment include fine-grained resource sharing and the consequent economic saving.

2.1.2 Server-less computing

Today's trends bring us the execution model known as *Serverless computing*: in this model, the cloud provider allocates resources on-demand, taking care of the server on behalf of their customer. In the serverless paradigm, developers of serverless applications do not handle capacity planning, configuration, and management of neither VMs nor containers.

Vendors of those services offer computing facilities to handle the code from the developer under the function-as-a-service pattern.

Fig. 2.1 summarize the architectural evolution from bare metal to server-less computing exposing how the IT research went from the design of several layers of virtualization abstractions to reduce them in order to achieve resource sharing and exploitation better.

2.1.3 An eternal return context

As Nietzsche discussed human life, computer engineering is also affected by the "Eternal return" principle: last years are affected by the return to historical patterns for IT environments. With the arising of Cloud and Edge computing, the distance between user terminals device and the actual hardware that executes jobs is going back to be higher, sometimes remembering mainframes times: Edge computing is enabling offloading of compute-intensive jobs from end-user devices for several reasons like increasing energy-saving; Amazon WorkSpaces [36] is a cloud-native persistent desktop virtualization solution that lets users access data, applications, and resources "as they go": advantages lay from increasing

agility and optimization of costs to a better management of the environmental pollution. Users will not need to renew their hardware time by time: they will use only the resources they need to rent them from cloud providers. A Key and straightforward use case is 3D design and rendering: spikes of resources requirements can be assessed by vertical scaling of those persistent desktop virtual machines, while most of the time, when resources required are back to normal, others can use them.

Finally, to keep in topic with the -as-a-service computing architectures, bare metal as a service is being offered by now from multiple cloud service providers like AWS and Google. Despite historically offered solutions seem to return in trends, new challenges arise:

- User experience given by solutions based both on Edge Computing and Cloud-Native persistent desktops are too affected by network latencies;

- The distribution of workloads across multiple clouds and Edge networks make it important to assess properly security strategies in several scenarios;

- Despite serverless computing seem to maximize resource utilization and to minimize infrastructure configuration, functions initialization times become key performance indicators that enable new objectives for scheduling strategies;

- Locality awareness has ever been important both in the design of the data-centers hardware infrastructure and in the design of components of applications; impact of the network become even too challenging when employing solutions like *memory-as-a-service* [18, 37, 38]

- multi-cloud and Edge computing environments need to handle replication and consistency strategies at the business layer, especially for stateful applications;

2.2 Orchestrators

The outlined trends indicate that container clustering is a solution for production environments. Kubernetes is an open-source cluster manager for Docker containers by Google, derived from Borg [39]. It outperformed Docker Swarm (the base tool integrated with the Docker core), with features such as auto-replication and auto-placement.

Mesos [40] is another promising open-source cluster manager. It is based on a distributed two-level scheduling architecture. At the lower layer, a Mesos master collects information on the free resources offered by the slaves and proposes, through the Dominant Resource Fairness (DRF) [41], the list of these available resources to the application-specific scheduling frameworks, allowing them to accept the resources and map the application components.

Mesos improves the use of the resources grouped on clustered slaves. Any Linux program can be run on a Mesos framework, characterized by an ad-hoc scheduling policy.

While Mesos is thought for generic use by exploiting its two-level scheduler for a multitude kinds of applications, Kubernetes seems to be the de-facto standard being provided by commercial cloud vendors. Without lack of generality, this thesis makes use of the abstractions typical of Kubernetes environments to the end of being synthetic when discussing patterns that have been integrated into the most important technologies driving the current landscape of cloud-native solutions.

2.3 Container resource allocation

Generally, resource allocation consists in the assignment of the available resources in very different scenarios: e.g., economics, human resources, delivery services and, last but not least, computing, networking and distributed systems.

This section explores resource allocation in the field of containers that have to be scheduled in multiple hosts environments.

Note that, usually, the software responsible for allocation of resources and placement of tasks in a distributed environment is referred as scheduler.

Works in literature use strategies able to work online, that is schedule tasks in the system and then re-adapt them at runtime, either using an event-based algorithm [42] or a time-based one [43]. Others only rely on offline scheduling, letting the work of guaranteeing run-time quality of service to other components as the current Kubernetes and Mesos schedulers [44, 45].

Finally, objectives can be wide and different across the available schedulers algorithms: challenges arise when one wants to take into account multiple real-world issues related to the SLAs of applications being deployed in a distributed system.

Problems in these scenarios easily turn into NP-completeness: for this reason, a huge gap is observed between proposed solutions in literature, and the ones employed in production systems like Kubernetes.

The container re-balancing mechanism in [42] proposes a resource-aware placement scheme to boost the performance of a heterogeneous cluster through proactive-optimization based on rapid live migrations, preparing the system for future workloads, especially on busy clusters, and minimizing the interference with the main scheduler. The authors of [46] propose a way to improve the standard scheduler for Docker

Swarm (Spread) based on the Ant Colony Optimization algorithm with a focus on resource balancing spreading the containers of a cluster by guaranteeing fair usage of resources on hosts.

In [43], the authors explore the problem of how effectively manage CPU utilization when many containers share a single set of resources. They work on Docker introducing a strategy based on time-slicing: with their mechanism, only a single container with all its threads and processes is scheduled at any time on shared resources thus minimizing the effects of oversubscribing the resources.

Complexity of the scheduler also arises when considering heterogeneous clusters. DRAPS [47] is an approach for scheduling on heterogeneous Docker clusters to pursue balancing of resources like CPU and RAM: the authors propose both an offline placement strategy and an algorithm to migrate containers whenever the monitoring system detects new bottlenecks on the worker nodes.

The authors in [46] and [47] add components to the Docker architecture but also significantly need to modify the orchestrator they refer to (Docker SwarmKit) in order to achieve the improvements requested, due to the monolithic design of Docker. Mesos and Kubernetes allow to overcome the effort due to the maintainance of forks of the orchestrators, respectively, through a two-level scheduling architecture and the operator pattern.

An interesting contribution exploiting Mesos is in Electron [48]. It is a framework aiming to reduce power peaks and energy consumption for the slave nodes in heterogeneous clusters. The authors adopt the standard strategies first-fit and bin-packing [49] together with external tools for power capping as *RAPL* [50] and a custom algorithm, *Extrema*, to provide software and dynamic power capping on slave nodes.

However, this solution focuses exclusively on enabling the power capping on the cluster nodes while avoiding to affect the execution time strongly. Electron objectives are neither to optimize resources utilization nor to provide guaratees for SLAs between providers and customers.

Most of the schedulers presented till now can be considered general purpose. However, quality of service requirements can vary based on the kind of applications being orchestrated.

The work presented in [51] proposes a container orchestrator dedicated to scalable data analytics frameworks. The aim is to improve the execution time of jobs of different data analysis frameworks on shared clusters. The paper focuses on the closeness between the containers, in terms of network hops and on the data locality, in order to put the processing containers close to the input data sources because of the requirements for data analytics frameworks. In particular, this work takes into account Hadoop/YARN and Apache Flink.

This work could have been integrated with the design pattern leveraged by Mesos which hosts multiple frameworks running concurrently on resources offered by the low-level scheduler based on DRF, letting them to accept or drop those offers in their second level scheduling system. However, the high-level frameworks should be taken into account as reported in [52].

In [53] the authors propose a job scheduling strategy for containers based on linear programming model where the objective function takes into account the container image transition costs from the image registry to the container host, the energy cost of the container hosts and the workload network transition costs from the end-clients to the container hosts. They compare their strategy with a binpacking [54] scheduler and a random scheduler. The modeling through linear programming is a well-

known way to better focus the requirements and objectives to consider while designing a new scheduling strategy.

However, an important focus, in the context of placement of containers, should also be given on the time needed to find a solution for a specific problem to reach its objective. While describing guidelines for rapidly scaling video application on Amazon EKS clusters [55], just as an example, authors show how spikes in traffic can led to scale pods in their cluster over 100%, so that in a 1-minute window the number of pods arise from 42 to 117. Orchestrators should have ability to appropriately allocate resources in a timely-manner.

Game theory is used in [56] to allocate resources between tasks submitted by customers. Recent literature exists on cache allocation, in which the NO allocates memory to third party SPs to minimize bandwidth consumption [57] or QoS and fairness [58] (CPU is ignored). Authors of [59] assume users send a sequence of tasks and each can run under different configurations, requiring a combination of different resource types. They assume users want to run as many tasks as possible and their utility is number of tasks run.

The Fuzzy Inference System (FIS) in [60] addresses to dynamically predict the most proper node where the given containers will be deployed. The most proper node is chosen as the least busy one. The authors consider different resource usage parameters taken by the */proc* file-system in the cluster's hosts and they use the collected information as the input of a FIS for which the output is a numerical value that represents the degree at which the node is busy. Moreover, they consider different FIS according to the specific container type (e.g. CPU-intensive, memory-intensive, etc..). Then the numeric values (output of the FIS) of each node is used to determine the best suitable node to deploy a container.

In [61] the authors design and evaluate DeepRM, a simple multi-resource cluster scheduler. They consider an online setting where jobs arrive dynamically and cannot be preempted once scheduled. Their objective function goes through the optimization of the completion time of the scheduled jobs employing a standard policy gradient reinforcement learning algorithm. However in the work, the authors consider a single large pool of resources, ignoring machine fragmentation effects.

As for the information to support the allocation decision, most work is based on a "monitor-and-decide" approach [60–62], but it is becoming common to also use detailed information on the workloads by the users [60].

Actually, providing both performance isolation and acceptable levels of QoS for cloud applications remains a challenge, especially in the case of network sensitive workloads. Statically partitioning resources and dedicating them to any service increments the costs of cloud provision and may create a big degree of under-utilization. Current solutions outside of academia are mostly implemented exploiting the concepts of resources constraints ("limits" and "requests" in Kuberentes [63]) with the aid of schedulers based on "filter, score and sort" strategies, e.g., the Kubernetes scheduler [64].

The network resources deserve a separate discussion. In many cases the overall QoS of distributed applications is affected by network performance interferences. They occur when containers do not get the required amount of bandwidth due to excessive network bandwidth usage by others. Generally, PaaS orchestrators give no guarantee for network performance. This makes most of the workflows deployed on the cloud susceptible to the high variability and unpredictability of the network performance [18, 65].

The main techniques for the network performance isolation provisioning are based on the flow control of the amount of data exchanged among the communicating entities.

2.3.1 Approaches to network isolation

Approaches to network isolation proposed in literature are spread across each layer of the OSI stack.

One of the first approaches was Quantized Congestion Notification, QCN [66], a Layer 2 congestion management algorithm developed for the IEEE 802.1Qa standard [67], to provide end-to-end congestion management in switched Ethernet. Afterwards, the same authors propose AF-QCN [68], an algorithm that adds a programmable bandwidth partitioning component based on AFD to the QCN Congestion Point mechanism. However, both the solutions (QCN and AF-ACN) refer to the layer 2 of the stack and require modifications of the network stack for each communication entity.

A similar approach, at the transport layer, is the focus of Seawall [17]. It is a system for network performance isolation in clouds. Using ad-hoc TCP tunnels among communicating VMs, it grants the ability to control the data flow and, as a consequence, force a fair allocation of the network capacities among co-hosted VMs. An important characteristic is its robustness against malicious users who deploy VMs for generating noise workload. However, Seawall offers no performance predictability and it also requires modifications over the stack of OS guest.

On the other hand, NBWGuard [69], is a design for network bandwidth management on Kubernetes. It lets Kubernetes manage network bandwidth as a resource (like CPU or memory capacity) while still using plugins for realizing the network specification desired by users.

2.4 Edge-Cloud offloading

Since literature on Edge Computing (EC) is vast, this work just focuses on work concerning resource allocation.

This field hasn't yet been investigated in depth.

This thesis will describe contribution in next chapters on the scenarios of Metro Edge Cloud and Mobile Edge Computing [26, 70], in which there are computation nodes concentrated in small data-centers located into the Network Operator Central Office (CO) or co-located in the base station.

EC is complementary to Cloud, i.e., the usual assumption is that a part of service computation is peformed at the Edge and the rest on the Cloud and similarly a part of the required data seats at the Edge and the rest on the Cloud. In a sense, the Edge-Cloud infrastructure is hierarchical [25, 71, 72], where Edge resources are the leaves, and the upper nodes are Cloud clusters. In [25] the decision of how much capacity must be provisioned across the levels of the hierarchy. In the following, a few challenges arisen from the deployment of 5G networks and Edge Computing facilities.

Multi-Tenant Resource Allocation Resource allocation among third party tenants is currently done in Cloud computing via pricing. However, we can consider infinite resource in the Cloud, so that they can be granted as long as the tenant is willing to pay.

At the Edge, instead, resources are limited and the NO, which owns them, want to allocate them in order to increase its own utility. Allocation of finite resources among different service providers (tenants), which compete for their consumption, has not vastly been explored in the context of EC. Some examples of this kind of problems can be found

in [57] and [58], where resource is mono-dimensional (storage), and the utility is the bandwidth reduction, QoS and fairness.

Resource Provisioning There is agreement that Edge and Cloud computing form a unique pool of resources, organized, hierarchically, and services can use both simultaneously. In this context, there is vast literature in resource provisioning, which regards the decision of how much resource should be deployed at the Edge nodes or at the Cloud [25, 73].

Network Slicing Network slicing consists in creating virtual network slices on top of a physical network infrastructure, whose owner has to allocate resources among slices. In this context, resources are mainly mono-dimensional (bandwidth [74]), with some exception ([56, 75] consider also CPU).

Service Adaptability Some work assumes that services can run under different configuration, thus adapting to the resources provided. Services span from Federated Machine Learning [76] to video streaming [77]. Other work [59, 78] assumes that multiple configurations result in different multi-dimensional resource usage and different QoE. However, most of this work, considers one only tenant.

Resource allocation for container-based EC The micro-service architecture is particularly suitable for resource allocation, as services can adapt to the resources available by launching/destroying the containers hosting micro-services [43, 53, 60].

2.5 Towards AIOps

In a way, AIOps can be considered a new framework on which academics and engineers should capitalize the effort achieved during the last 20 years studying and actuating strategies to optimise workloads deployed in Clouds, Fog and Edge computing.

AIOps is expected to be production-grade for 80% of companies by 2024 [30]. It will consist on multiple achievements going from autonomous service support ticketing system to better strategies employed in orchestrators of PaaS environments on which Cloud-native applications (CNAs) are first citizens and complete knowledge of the system is given by -as-a-code patterns. In this section and in the whole thesis, the focus on AIOps will be related to the orchestration of CNAs and to autonomous strategies for guaranteeing SLAs.

Cloud-native applications CNAs are essentially applications resilient to cloud outages, and which can scale on a fine-granular basis. They are built as a network of independent services communicating over standardized interfaces. Works in literature go from software design principles [79] to strategies to guarantee SLAs in scenarios like Biomedical [80] and 5G [3, 4], exploring a vast set of optimization properties as energy [81], network-latency [1], cost [82]...

Methodologies to model CNAs can involve graph theory [83], sandboxing [84], queueing theory [85], or n-gram dictionaries by log parsing [86]; objectives go towards micro-services capacity [84], costs [84], processing time [85] etc...

Monitoring As anticipated, most work in literature is based on a "monitor-and-decide" approach. Monitoring can be distinguished into black-box and white-box, based on the knowledge of the monitor about

the applications. Different works involve passive monitoring for networks [87] and cloud services [88] or evaluate active-passive monitor differences [89]. Moreover, companies like Splunk [90] and Sysdig [91], are actively working to provide AIOps-oriented monitoring for CNAs.

SLAs, CNAs orchestration and AIOps Objectives of optimization in the cloud go towards micro-services capacity [84] planning, costs [84], processing time [85], auto-scaling [92].

SLAs and QoS are strictly related through the SLIs and the SLOs defined in these agreements. SLOs stipulate performance goals for cloud applications, microservices, and infrastructure: as a consequence, the monitoring of CNAs and their underlying infrastructure is crucial to (i) execute the proper task in-time when anomalies occur, and (ii) produce highly detailed reports in case of complex disputes due to service unavailability or damages to the application providers or the resource providers.

In [14, 93], authors aim to cover the gap between the lower-level metrics obtained by monitoring systems with the high-level SLOs stated in the SLAs, and provide a taxonomy of the most valuable SLOs for cloud applications to compare them in terms of workloads and performance goals. Other works tend to find systematic strategies to allow an efficient design of SLAs both from a business and IT engineering perspective, through tools [94, 95], decision support strategies [15] to minimise penalty payments through statistical models, or analytical strategies to model [96] SLAs as ontologies that provide facilities to capture semantics of those agreements and avoidance of interoperability issues.

Autonomous scaling of CNAs In the context of Cloud computing and AIOps, auto-scaling mechanisms are the most frequently used solutions to satisfy QoS properties and SLAs guarantees assuring an efficient use of resources at low cost for both the resource providers and customers.

Different research contributions explore the elasticity of CNAs in Clouds and provide different strategies to achieve efficient scaling of resources just in time to minimise number of violation of the SLAs; however, the challenges in this field are broad and the gap between research proposals and the available production-grade strategies is still very large. This field is explored from different perspectives: authors of [9, 97] leverage strategies based on time-series forecasting, respectively, using ARIMA models and Facebook Prophet [33]. Others solutions are related to efficient task allocation leveraging deep reinforcement learning [98] or multi-cloud containers placement with reinforcement learning strategies to choose between vertical or horizontal scaling [99]. More recently, on the 5G and Edge Computing facilities, authors explore auto-scaling of virtual network functions (NFV) both in the case of multi-domain and multi-tenant Edge environments [100], and in the case of Cloud-edge offloading, by exploiting the concepts of service elasticity [3, 4].

Even though most of the solution to achieve spikes in resource usage is based on horizontal scaling, the main contribution of [62] is to present the ability of ElasticDocker to power the vertical elasticity of Docker containers in a way based on the IBM MAPE-K principles.

ElasticDocker is able to scale up and down Docker containers when the application workload changes with an approach based on modification of resource limits directly in the Linux control groups associated to the Docker containers. By using capabilities of CRI-U [101], this strategy also provides live migration of the containers whenever the required resources start overloading their host. In our work there are no ways to achieve at the vertical scalability, which could be unpractical in the resource-constrained scenario of Edge Computing.

Network adaptiveness strategies Software Defined Networks (SDN) are widely employed today in production environments spanning from

SDWANs of network operators [16], to inter-datacenter wide area networks, as Google's B4 [102], up to internal networks in platform as a service deployments [37]. Along with the rapid development of networks and clouds, new operational scenarios emerge: low latency challenges arise in the fields of games, virtual reality and automotive engineering [103]. Authors of [104] explore strategies to provide high-quality networks for mobile clients of Virtual Reality environments. In scenarios like Video Content delivery services, authors of Pensieve [105] employ neural networks and reinforcement learning to design ABR video streaming algorithms that adapt to a wide range of environments with the aim of maximising quality of experience for users. The authors, in [106–108], propose an algorithm, Alienated Ant Colony Optimization, to handle, at a lower-level, routing rules of SDNs aiming to optimise throughput, latencies and energy, through a vendor-agnostic orchestrator, jFlowlight.

Given the impact of network on modern, distributed, CNAs [37], research challenges in the field of network optimization arise day by day, especially while the level of hierarchy has started, reducing as anticipated in the previous chapter, with a focus that passed from VMs, to containers, up to functions. In those scenarios, strategies optimising network isolation in terms of security and network routing as in [106–108] to handle steady impact of network issues, but especially spikes in traffic that can lead to violations of SLOs. As from Covid-19 outbreak, interest of academia is also turning back to the management, e.g., previoinionig and re-provinoing, of resources during disruptive events.

Part II

Scheduling containers in cloud and edge computing

Chapter 3

Introduction

« Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice, "without pictures or conversations?" »

Lewis Carrol

An effective mapping between services and resources is a critical issue in cloud environments [109, 110]. The current implementations of platform-as-a-service, PaaS (e.g., Openshift [111]) offer basic QoS-aware functionalities to guarantee a certain degree of performance for hosted applications. In this part, two complementary works are described on the field of the placement of containers in two different scenarios:

- generic applications in heterogeneous PaaS clouds with network optimization objective;

- applications leveraging *service elasticity* to optimise the utility a network operator can obtain, providing computing at the Edge for service providers.

Communication-aware placement of containers in PaaS. In the environment of containerized applications, it is possible to select for each container the resource requests and limits [63], in terms of the number of CPU shares, the amount of main memory, the number of IOPS, the bandwidth [1, 17], the quantity of storage...

This approach represents the one mainly adopted in real-world scenarios (Amazon [112], RackSpace [113], Azure [114], Google Cloud [115]), and suffers of three main issues:

1. the impossibility to guarantee *performance isolation* when several virtual machines (VMs) are running on the same multi-core machine, and when multiple containers run within the same VM [116];

2. the impossibility for the applications' architects to precisely evaluate the performance of the interacting distributed components, deployed as containers on different VMs;

3. The difficulty to relate the performance of any application to the hardware resources allocated for its components at run-time: hosts are agnostic about the tasks they are executing.

To tackle the first issue, the providers should enact deployment strategies to reduce concurrency on the same resources (e.g., minimizing the number of components sharing the same computing and networking resources). Cooperation between customers and providers is needed to overcome the other two issues: providers will be able to deploy applications effectively only when customers provide an adequate description of them [1, 5] (in terms of the number of components, relationships among them, etc.).

Chapter 4 presents a strategy that combines (i) the exploitation of the inherent flexibility of virtualization and containerization, and

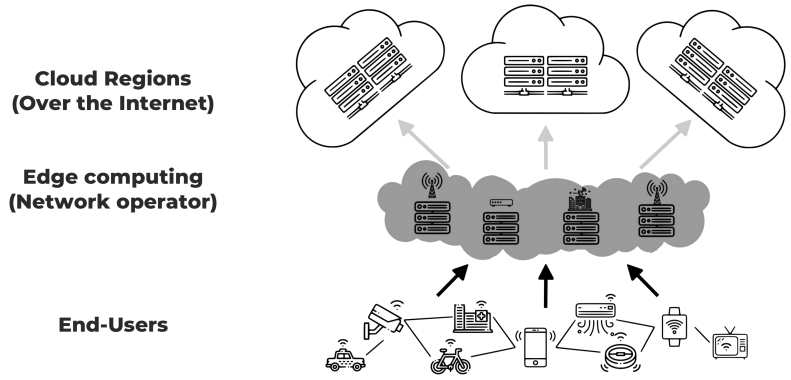

Figure 3.1: Cloud and Edge computing

(ii) the availability of information about the structure of an application; this information is formalized into two specific parameters: the *closeness* among the hosts of the application similar to what was introduced in [117], and the *isolation index*, proposed to measure the degree of mutual influence among components of a cloud-native application. The approach aims to minimize both the mutual interference between the applications and the number of costly run-time adaptation procedures (e.g., micro-services migration); it represents an effective deployment strategy that enhances the predictability of applications behavior and reduces the number of run-time adaptation procedures to meet the Service-Level Agreements (SLAs) for applications.

Cloud-Edge offloading and location-aware placement of containers at the Edge. Under the arising paradigm of Edge Computing (EC), computational capabilities, e.g., memory and processing elements, are deployed directly in the access networks, close to the users. It enables low latency applications, reduces the traffic going out from the access networks, and improves user experience. EC, from this point of

view, is complementary to Cloud [25]. The usual assumption is that some service computation is performed at the Edge and the rest on the Cloud. Similarly, a part of the required data seats at the Edge and the rest on the Cloud.

In the Cloud, resources are usually assumed *elastic*, i.e., they are always available, as long the third-party Service Provider (SP) is willing to pay. On the contrary, contention emerges in the Edge between SPs sharing limited resources, and the problem arises of how to allocate them between SPs.

Most of the work [56,61,71,72] models Cloud and Edge resources under a *task-oriented* viewpoint. , i.e., as in the grid computing era, where jobs are composed by tasks that consume resources and terminate. By adopting a *service-oriented* viewpoint, the problem of allocating containers in the edge to make computation closer to the end-user is similar to the well-known problem of leveraging multi-cloud architecture and federation of clouds to put services closer to the location on which they are requested, as in the usual multiple regions or multiple availability zones scenarios.

However, in the case of Edge computing, other requirements arise:

- Networks cannot be considered safe and several networking constraints have to be addressed;
- Interests of the stake-holders are different: a service provider would improve the quality of experience for its end-users, while the network provider wants to reduce the amount of inter-domain traffic paid to make the users able to reach requested services;
- The etherogeneity of network providers facilities does not allow yet an actual implementation that is open directly to the service providers.

In chapter 5, a Network Operator (NO), owning limited computational resources in its Edge network, must decide how to distribute them to different Service Providers (SPs). The goal of the NO is to maximize its utility, which can represent bandwidth, operational cost saving, or improved experience for its users [57, 77].

The two works are complementary since the first one focuses on the communication that happens internally to a cluster of machines hosting containers, while the latter is related to the maximization of the utility that a network operator can obtain by allocating service providers' application in its network. In a way, combining the two works means to exploit optimal internal network configurations (chap. 4) with minimal inter-domain traffic. Whether the scenarios lay both in cloud/edge offloading setups or in multi-cloud environments, they open the way for a cloud of network optimized workflows.

Chapter 4

Communication-intensive applications in PaaS clouds

Let's consider a set of physical machines (PMs) hosting different VMs and sharing multiple kinds of communication channels.

The performance of these communication channels depends on the geographical location of such PMs: e.g., the ones hosted on the same rack can communicate over a 10GbE channel through only one L2 switch; instead, machines of different racks can communicate over a shared 1GbE link.

Each VM encompasses a set of resources, typically: (i) a given number of cores; (ii) a specific amount of main memory; (iii) one or more virtual communication channels.

The whole set of VMs managed by the same orchestrator will be referred to as *system*.

Hereinafter, they will be considered Cloud-native applications consisting of containerized micro-services. For their deployment we refer to the abstraction layers provided by Kubernetes [118] and Redhat Open-

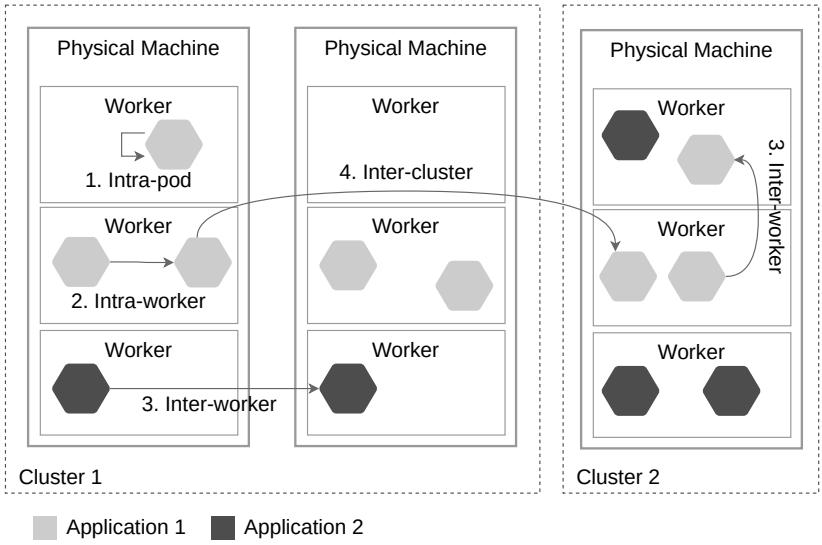

Figure 4.1: A simple reference scenario

shift [111] that are real-word examples of orchestrators to manage PaaS solutions. In this scenario, each micro-service can run in a *pod* (the atomic deployment unit in Kubernetes consisting of a set of one or more, strictly coupled, containers) and each pod can run on only one VM, henceforth called *worker*, at a time.

Figure 4.1 shows a cloud system consisting of nine VMs hosted on three PMs belonging to two clusters. Two applications are also shown as consisting of multiple pods deployed into the workers.

In this scenario, four types of communication modes can be considered:

1. Intra-pod: containers running into the same pod share the same network channels. This work ignores this kind of communication because our atomic unit of deployment is the Pod itself.

2. Intra-worker communication: Pods hosted on the same worker will communicate over the virtual communication channels of the worker itself.

3. Inter-worker communication: pods deployed on different workers communicate over an overlay network. This overlay network can either be virtual, if the workers are located into the same physical machine, or physical, when the workers are hosted into different machines.

4. Inter-cluster communication: when workers belong to different clusters, their communication performance depend on the underlying network which generally consists of heterogeneous channels, e.g., in terms of different bandwidth and latency.

The following sections will introduce two parameters to represent the communication between micro-services of an application (the *isolation index*) and the communication between all the workers (the *closeness*). These two parameters will be combined into another one, the *flow index*. Maximizing the flow index will lead to guarantee a better QoS.

4.1 Allocation Issues in Clouds

For cloud-native applications, a key issue is the careful arrangement of a large number of micro-services over a huge pool of heterogeneous shared resources and effectively executing them in parallel.

The orchestrators should allocate the adequate amount of the resources they manage to the pods and apply suitable scheduling policies to guarantee adequate application performance.

Table 4.1: Summary of notation

General notation	
A	Generic cloud-native application
$\mathcal{G}_A$	Graph of the application A
$\mathcal{V}_A$	Set of components of the application A
$\mathcal{E}_A$	Set of communication links (edges) of the application A
w_A	Adjacency matrix representing communication rates of the application A
r_A	Matrix representing resources required by components of A (Vertex labels)
$\mathcal{M}$	Set of workers
$\mathcal{M}_A$	Set of workers available to deploy compoents of A
$m \in \mathcal{M}$	Generic worker
H	Number of workers used to deploy A
$\mathcal{V}_A^{m_h} \subseteq \mathcal{V}_A$	Partition of $\mathcal{V}_A$ consisting of all components hosted on worker $m_h \in \mathcal{M}_A \subseteq \mathcal{M}$
$\kappa_{m,n} = 1/d_{m,n}$	Closeness between workers pair $(m,n) \in \mathcal{M}^2$
$W_{m,n}$	Sum of all the weights of links between components hosted on workers m and n
IQP/MIP formulation	
X^A	number of components for the generic application A
$w_{j,k}^A$	element of adjacency matrix w_A representing communication rate between components $(j,k) \in \mathcal{E}_A \subseteq \mathcal{V}_A^2$
$r_{j,l}^A$	the amount of resource l required by the component $j \in \mathcal{V}_A$ (element of r^A)
M	number of workers ($\lvert\mathcal{M}\rvert$)
L	number of resource types (e.g.: RAM, CPU, Storage...)
$c_{m,l}$	Amount of resource type l available on worker $m \in \mathcal{M}$

The focus is on devising flexible strategies for managing these resources and optimizing their utilization. This often implies maximizing resource sharing between the hosted components.

Admittedly, the interference, which often affects the use of the shared resources, could hinder the performance isolation among the pods, making the orchestrator unable to accurately predict the behaviour of an application and guarantee the desired QoS. However, reasonable prediction capabilities should still be possible under the following two conditions:

- The number of pods running in parallel on the same worker have to be as few as possible in order to lessen the interference effect; a large number of pods running on the same worker could adversely affect the performance of hosted applications due to the competition for the underlying physical resources, e.g., memory interference effect or shared I/O operation on the same disk [119].
- Communication between pods hosted on different workers should be minimal; when pods hosted by workers belonging to different PMs interact, their performance strongly depends on external factors (e.g., network delay, bandwidth decrease, network fault, etc) out of the direct control of the orchestrator.

The first of the above conditions can be easily satisfied by limiting the maximum number of pods which can be hosted on the same worker; this avoids (or at least strongly reduces) resources interference. Many orchestrators, like Openshift, are able to setup proper policy rules to limit the maximum number of pods running in a worker.

Satisfying the second condition is, instead, more difficult. As stated before, many cloud applications consist of several collaborative distributed components, exchanging messages in order to coordinate their behaviour and reach a common goal. The lack of direct control over the communication channels among the physical resources on which these micro-services

run makes their performance unpredictable, especially for Communication-Intensive Applications [120].

The greater the traffic on the shared network, the greater the degree of uncertainty introduced in the performance of applications (see Section 4.2 for more details).

Performance isolation for a set of cooperating micro-services could be obtained by using private communication channels, i.e., channels accessible exclusively by these micro-services. This solution, however, strongly limits the resource exploitation: reserving a portion of the network resources only to a small subset of the applications adversely affects the performance of the entire system. Different solutions are being proposed in literature to optimally overcome this issue in Cloud [69, 121].

The combined use of multi-core together with virtualization technologies gives rise to a class of high-speed, low-latency, reliable and robust communication channels among the cores and, as a consequence, among the processes (i.e. containers in pods) running in the same worker. As the authors will analyse in more detail below, the performance of an application may be better assessed and controlled by:

- properly driving both the internal (among pods on the same worker) and external (among workers on different PMs) physical communication channels;

- deploying the pods that generate most of the communication traffic either on the same worker, whenever possible, or on workers close to each other, in order to limit external interferences.

The following sections will show how suitable deployment strategies may be based on performance isolation and closeness.

4.1.1 Application models

In this work, a generic cloud-native application A is represented by a weighted and vertex-labeled graph, called Application Graph in the following:

$$\mathcal{G}_A = (\mathcal{V}_A, \mathcal{E}_A, w_A, r_A) \tag{4.1}$$

where:

- $\mathcal{V}_A$ is the set of nodes representing the pods of the application A;
- $\mathcal{E}_A$ is the set of edges representing the communication relationships between interacting pods;
- w_A is the adjacency matrix: each element $w_{i,j}$ denotes the communication load in terms of rate of information to be exchanged between pods i and j of the application A. These weights can be either *time-varying* or *constant*;
- r_A is a matrix which comprises the resources requirements of an applications' pods: given a resource type l (e.g., memory), $r_A^{i,l}$ represents the amount of resource l required by pod i of application A. This matrix represents the labels of the graph vertices.

Given the functional requirements defined above, non functional requirements of the application A, i.e., the service-level objectives (SLOs) stated in a SLA [122], can be represented, without loss of generality, as a set SLO_A of $<$ *entity*, *predicate* $>$ pairs, where *entity* can be either a vertex or edge in $\mathcal{G}_A$ and *predicate* is a required property: e.g., $replicas = 3$, $response\_time \leq 300ms$ in 90% of the requests;

The ability to enact effective resource allocation and reservation policies for an application A depends on the knowledge (and the accuracy of

the information) modeled with the application graph $\mathcal{G}_A$ and the predicates in SLO_A.

However, in real-world scenarios, information about $\mathcal{G}_A$ may lack of completeness and accuracy. As it will be shown in sec 4.2, the more the information about the graph, the more effective the mapping strategy of components onto workers.

4.1.2 Definition of Isolation Index

To deploy an application A, an orchestrator has to schedule each pod of A on a worker endowed with the appropriate amount of resources. Letting $\mathcal{M}$ denote the set of workers managed by the orchestrator and $\mathcal{A}$ the set of autonomous applications that it has to deploy.

The orchestrator has to map the pods of all the applications according to a function:

$$\text{map} : \bigcup_{A \in \mathcal{A}} \mathcal{V}_A \times Time \to \mathcal{M} \tag{4.2}$$

The mapping, for any application $A \in \mathcal{A}$, can be represented as the restriction:

$$\text{map}|_A : \mathcal{V}_A \times Time \to \mathcal{M}_A \subseteq \mathcal{M} \tag{4.3}$$

under the resource requirements defined by the matrix r_A and the non-functional requirements SLO_A. In eq. (4.3), $\mathcal{M}_A$ denotes the set of all workers available at any time to deploy pods of A.

For each pod $v \in \mathcal{V}_A$, $\text{map}|_A(v, t)$ is the worker where v is deployed at time t;

Indeed, any worker $m \in \mathcal{M}_A$ can simultaneously host none, one, or more than one, pods (whether belonging to A or another, even someone else's, application).

The restriction $\text{map}|_A$ is surjective and, thus, right-invertible. We will denote as $\text{hosted\_on}|_A : \mathcal{M}_A \times Time \rightarrow \mathcal{V}_A$ the inverse image of $\text{map}|_A$:

$$\text{hosted\_on}|_A(m,t) = \{v \in \mathcal{V}_A \mid \text{map}|_A(v,t) = m\} \quad (4.4)$$

Clearly, $\text{hosted\_on}|_A(m,t)$ is the set of pods hosted on $m \in \mathcal{M}_A$ under mapping $\text{map}|_A$ at time t. $\text{hosted\_on}|_A(m,t)$ can be the empty set, a singleton or a set with multiple pods.

Now, the mapping process defined for application A by a given $\text{map}|_A(\cdot,t)$ induces, at any time t, a partition of the set $\mathcal{V}_A$ of the application podsinto $H \leq |\mathcal{M}_A|$ disjoint sets $\mathcal{V}_A^{m_1}, \ldots, \mathcal{V}_A^{m_H}$ such that each $\mathcal{V}_A^{m_h}$ consists of all and only/exactly those components of $\mathcal{V}_A$ hosted on the same worker m_h, i.e., at time t, $\mathcal{V}_A^{m_h} = \text{hosted\_on}|_A(m_h,t)$.

Formally, this partition is defined, at time t, as:

$$Part_A(t) = \{\text{hosted\_on}|_A(m,t) \neq \varnothing \mid m \in \mathcal{M}_A\} \quad (4.5)$$

We will call any set $\mathcal{V}_A^m$ a *components cluster*, or simply a *c-cluster*, to highlight the fact that the components forming it get "clustered" within the same worker.

Recalling application A is represented by a graph $\mathcal{G}_A$, any partition defined by (4.5) induces H sub-graphs $\mathcal{G}_A^{m_1}, \ldots, \mathcal{G}_A^{m_H}$ of $\mathcal{G}_A$, where, for each $m \in \mathcal{M}_A$, $\mathcal{G}_A^m = (\mathcal{V}_A^m, \mathcal{E}_A^m, w_A^m, r_A^m)$.

Intuitively, $\mathcal{G}_A^m$ is obtained from $\mathcal{G}_A$, building $\mathcal{V}_A^m$ by deleting nodes placed outside the worker m, and $\mathcal{E}_A^m$ by pruning edges leading out of $\mathcal{V}_A^m$: $\mathcal{E}_A^m$ is referred to as *private* for the worker m, in that it collects communication links between the pods within the same worker m.

In general, the ratio between the total amount of information exchanged within a c-cluster and the total amount of information ex-

changed among pods of the application measures the isolation of that c-cluster. This is the *normalized isolation index*, and can be computed based on the application graph adjacency matrix, recalling that edge weights represent the interaction load between the pods of the application. Based on the work in [1, 5], a formal definition of the *normalized isolation index* is given in the following:

Definition 1. *Let us consider an application A defined by the graph $\mathcal{G}_A$ as in eq.* (4.1)*, a set of workers $\mathcal{M}_A$ and a mapping function* $\text{map}|_A$*. The normalized isolation index of A at any time t is the ratio:*

$$nii_A = \frac{\sum_{m \in \mathcal{M}_A} \sum_{(i,j) \in \mathcal{E}_A^m} w_{i,j}^m}{\sum_{(i,j) \in \mathcal{E}_A} w_{i,j}} \tag{4.6}$$

In (4.6), the numerator represents the weight of all interactions between the pods of the application running on the same worker, i.e. the rate of information flowing between pods through *private* channels; the denominator represents the total weight of all interactions between any pair of pods of the entire application A.

nii may vary over time and lies in the interval [0,1]. It is 1 when all the edges are private, which means that the overall communication takes place within a worker and never between workers. It is 0 when all the edges are not private, which means that the overall communication takes place between different workers and never between pods on the same worker. Maximizing *nii*, as defined in Def. 1 means grouping highly-coupled pods within the same worker insofar as possible.

An effective mapping for a single application should maximise the rate of information flowing across private channels, i.e., exchanged within a c-cluster. However, given a set of applications managed by a cloud

orchestrator, the weights of the internal communication for each one may be strongly differentiated (and thus incomparable) not only in terms of inter-operability patterns and technologies but especially for the amount of information exchanged. Normalization as in def. 1 hides this workload heterogeneity.

As an example, let us consider a set of two applications, A_1 and A_2, with a network load completely different; e.g., a 1Mbps throughput for application A_1 vs 1Gbps for application A_2. In this case, maximizing the isolation index as defined in Def. 1 for each application can lead to inadequate allocations of the network channels. Let us suppose to allocate applications according to the mapping a, such that $nii^a_{A_2} < nii^a_{A_1}$.

Def. 1 had not allowed the mapping a to compare the actual degree of communication load of the applications; an orchestrator should prefer to use a better mapping, b, deploying A_1 and A_2 so that $nii^b_{A_1} < nii^a_{A_2} < nii^b_{A_2} \leq 1$.

Mapping b cannot be implemented by an approach that maximizes either the normalized isolation index of all applications or the sum of them.

Looking forward to the orchestration of different applications featured by heterogeneous network loads, the following non-normalized version of the isolation index parameter can be adopted:

Definition 2. *Let us consider an application A defined by the graph $\mathcal{G}_A$ as in eq. (4.1), a set of workers $\mathcal{M}_A$ and a mapping function* $\text{map}|_A$. *The isolation index of A at any time t is given by:*

$$ii_A = \sum_{m \in \mathcal{M}_A} \sum_{(i,j) \in \mathcal{E}^m_A} w^m_{i,j} \tag{4.7}$$

Based on this non normalized definition, two new approaches can be taken into account: maximizing either (i) each isolation index of the given applications or (ii) the sum of them.

In the first case, the scheduling will improve the network QoS for the single hosted application.

In the second case, the orchestrator looks for optimizing the overall network resource utilization (e.g., saving energy or bandwidth or improving quality of the overall physical communication channels among all the involved workers).

In the next sections, this work will mainly focus on this second approach.

4.1.3 Closeness: notion and formal definition

In the previous section, the basis for efficient application deployment boils down to the placement of pods on the workers in such a way to minimize the ensuing communication costs.

Furthermore, when the communication flow among workers is elevated, their placement should be as *close* to each other as possible. This requirement can be conveyed through the notion of *distance*, or its converse *closeness*, among workers, that makes it possible to evaluate a network-oriented placement of Communication intensive applications.

The distance is viewed here as an abstraction of such physical quantities as latency, response time, bandwidth and the like. There is no need to provide any detail beyond the fact that distances are non-negative real numbers.

Actually, hosting a group of pods on the same worker can be considered a first level of clustering, the most effective one; anyway intra-worker communication is more effective than inter-worker communication, even when the latter takes place within a data-center, high-speed, LAN. How-

ever, the cost of inter-worker communication, even in the scope of a single cloud provider, may vary by several orders of magnitude depending on inter-worker distance; For instance, the AWS infrastructure is strongly sensitive to the way physical machines are spread through availability zones, whether or not they belong to the same AWS Region. The impact of distance on performance is even stronger when the workers hosting micro-services based applications are scattered among different cloud providers [123]. Therefore, taking into account the notion of worker *distance*, can reduce the network cost even more than just placing pods optimizing the *isolation index*.

Given $d_{m,n}$, the distance between two workers $m, n \in \mathcal{M}$, its inverse, the *closeness* $\kappa_{m,n}$ is:

$$\kappa_{m,n} = \frac{1}{d_{m,n}} \tag{4.8}$$

Endowed with a measure of how close workers are, any application described by the graph $\mathcal{G}_A$ of equation (4.1) must be placed accordingly.

Whatever the deployment strategy, modeled by the (4.2), the information flow between workers can be computed by adding up individual flows of edges between pods belonging to different workers $m \in \mathcal{M}$, for all applications $A \in \mathcal{A}$.

Consider two workers $m, n \in \mathcal{M}$. $W_{m,n}(t)$ is a measure of the amount of information flowing between the two workers:

$$W_{m,n}(t) = \sum_{u \in \text{hosted\_on}(m,t)} \sum_{v \in \text{hosted\_on}(n,t)} w_{u,v} \tag{4.9}$$

where $\text{hosted\_on}(\cdot,\cdot)$ is the inverse image of map in (4.2). The higher $W_{m,n}$, the more coupled m and n are.

Note that $W_{m,m}$ reduces to the inner sum in equations (4.6) and (4.7) and still represents the amount of private communication between components hosted on the same worker.

Leveraging the use of both the concepts of *isolation index* and *closeness* allows to define the *Flow Index*.

Definition 3. *Let us consider a set of workers, $\mathcal{M}$. The flow index of $\mathcal{M}$ at any time t is given by:*

$$\phi = \sum_{m,n \in \mathcal{M}} \kappa_{m,n} \cdot W_{m,n}(t) \tag{4.10}$$

where:

- *$\kappa_{m,n}$ is the closeness between workers m and n;*
- *$W_{m,n}(t)$ at time t, is the total flow between workers m and n as in (4.9).*

If $m = n$, as already stated above, $\kappa_{m,n} = 1$: this equation becomes identical to the *isolation index* defined in (4.7). Otherwise, $\kappa_{m,n} \ll 1$.

Maximizing (4.10) amounts to ensure that (i) the highly-coupled pods are placed either on the same worker or on workers as close is possible to each other, and (ii) that workers close to each other are those that communicate more intensely.

4.2 Mapping strategies

The effectiveness of a mapping strategy can be strongly enhanced by the available information on the application graph $\mathcal{G}_A$. Based the thoroughness of this knowledge, three different application model are considered:

1. *Set of Components*: only the set $\mathcal{V}_A$ of the pods of the application is known; no information is available about its graph;

2. *Simple workflow*: the structure (i.e., edges $\mathcal{E}_A$) of the graph $\mathcal{G}_A$ is fixed and the values of weights w_A are constant in time;

3. *Timed workflow*: $\mathcal{G}_A$ can be time-variant.

These three different models provide increasing levels of knowledge about communication of pods within the application, and enable the adoption of suitable strategies for mapping the cloud-native application onto available resources.

The mapping strategy should also depend on the QoS profile of the application: when the application provider requires a strictly guaranteed SLA, the application has to be deployed and executed accordingly. Otherwise, if the SLA can be satisfied with "Best Effort" strategies, the application can be deployed and executed by simply trying to optimize the overall utilization of the managed resources. It is recalled here that the focus of this work points out an optimal strategy to achieve the second case: improving the QoS whenever the applications do not require QoS-guaranteed deployments.

4.2.1 Set of Components

In this case the only information available is the list of the pods of the application (and, optionally, their SLOs): no information about communication interactions is given. Under these conditions, it is not possible to optimize, *a priori*, the mapping; in order to maximize the flow index, a run-time, adaptive solution has to be chosen. To this aim, monitoring the podsof an application (as in [124]) allows to assess the communication load (e.g., in terms of frequency and quantity of exchanged messages) and reveal recurring communication patterns. The experience of a communication pattern solicit the system to migrate the involved pods in order to place them as close as possible maximizing ϕ: the pods will only be

actually migrated if the cost of the migration process is negligible respect to the utility in terms of communication load.

4.2.2 Simple Workflow

In this case, the submitted application is equipped with static information about interaction among its pods. This knowledge improves the placement of the pods on the available workers. If all the pods can be hosted on a single worker of the available pool, the mapping process will simply consist in identifying the best one. Instead, if the number of pods and their requirements are so that no single worker is adequate to host all of them, the mapping strategy becomes non-trivial.

As known in literature, mapping is a well-known, NP-complete problem, which makes searching for the optimal solution a highly time-consuming process [117, 125, 126]: heuristics are available either to optimize the placement of communication intensive applications within a Cloud [117], or to reduce the communication load over wide area Edge-Cloud networks [3, 4].

Moreover, the cloud is highly dynamic: creating, terminating and scaling up and down of applications imply that the chosen mapping configuration is optimal only for a limited time window. To mitigate this problem, it is necessary to either implement an admission control policy limiting the number of running applications, or reduce the number of physical resources allocated to the application. The first solution limits the system throughput, the last one impacts on the application performance. An effective solution can derive from the knowledge of the evolution of the communication among the workers: it can optimize the resource exploitation without affecting neither the application performance nor the system throughput.

Section 4.3 will refer to this case to provide an optimal strategy based on the IQP formulation according to section 4.1.

4.2.3 Timed Workflow

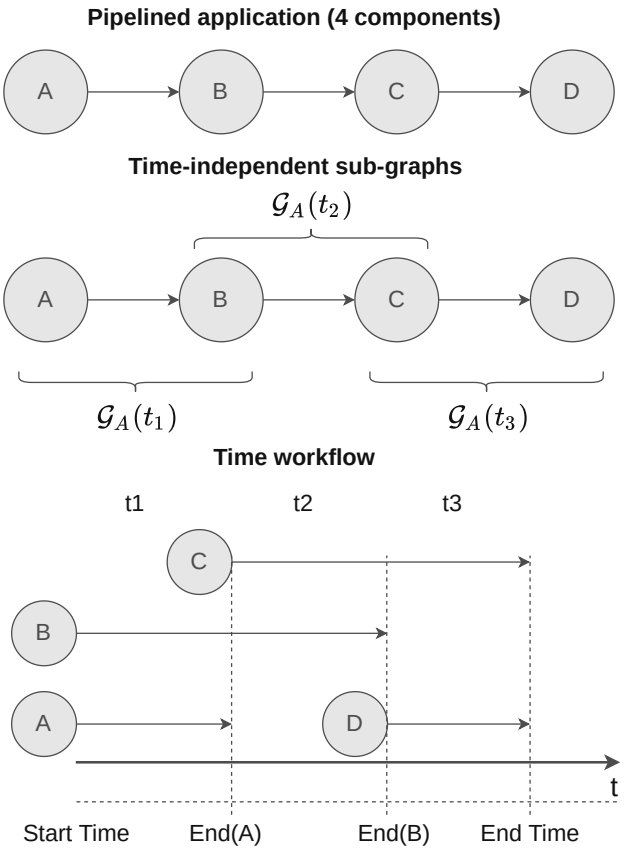

Figure 4.2: Information about application A

Listing 4.1 Time workflow yaml description for application A

```
1  # Time based description of a 4 components,
2  # pipelined, application
3
4  timeDescription:
5    - id: t1
6      components:
7        - A
8        - B
9      transaction:
10       event: End(A)
11   - id: t2
12     components:
13       - B
14       - C
15     transaction:
16       event: End(B)
17   - id: t3
18     components:
19       - C
20       - D
```

In some cases, application providers can provide a better description of their applications by introducing the temporal dimension on the elements of the w_A and r_A matrices (sec. 4.1.1).

In practice the workflow of these kind of applications can be traced back to discrete series of simple workflows, under the hypothesis that the system goes through steady states for the overall time (the epochs in [6]). Any *simple workflow* of the series will be valid for a time interval. The relevant weights, w_A could be approximated with the average value observed in the past histories or could be forecast based on other considerations of the expected behaviour of the application. Thus, within each epoch, edge weights have to be assumed to be constant (although they could actually slowly fluctuate).

This is a necessary condition to make the knowledge of $\mathcal{G}_A$ worthwhile for the application deployment Following this time splitting and exploiting the model in [6], a timed workflow can be partitioned into a sequence of n time-independent sub-graphs $\mathcal{G}_A(t)$.

In particular, the time in which an application runs, is split into a sequence of n time windows $(t_0, t_1, \ldots, t_n)$.

The application providers should specify a list of pods running at the same time and the *transition* event(s), i.e., the event or the group of events, that cause a transition from a time window to the next one.

As a consequence, the workflow of application A (represented by G_A) can be viewed as the sequential execution of the n sub-applications $A(t)$, for $t \in (t_0, t_1, \ldots, t_n)$.

Figure 4.2 shows the graph of a simple pipelined application, its time-based decomposition in sub-graphs, and the related *yaml* description into Listing 4.1. Different sub-graphs related to different epochs can exploit the same set of physical resources, at different times.

Since different time-independent applications A_i can be deployed on the same physical resources and, as a consequence, on the same communication channel(s), the probability of using the same channel, at a different time, for different communication, increases thus also the probability to render (near-)optimal ϕ.

The mapping process, in this case, consists of using, for each sub-graph, G_{A_i}, the same procedure described in the second scenario. The reduction of needed resources, for each time window, has a great impact on the performance related to the deployment of the application, because it increases the probability of finding *close resources* and reduces communication latency.

Even if this approach can be applied to many type of applications, it provides an effective improvement for long-running applications such us scientific workflow or big data-oriented analyses.

4.3 IQP Formulation

This section formulates, in an integer quadratic programming algorithm, the map$(\cdot,\cdot)$ function of eq. (4.2) optimizing the communication for a set of *simple workflow* applications.

Given N applications to be deployed on the Cloud *system*, the problem of maximizing ϕ can be represented by a *multiple quadratic multi-dimensional Knapsack* Problem (KP) [127].

"Multi-dimensional" are the constraints on the different types of involved resources.

"Multiple" is the availability of a number of workers on which the orchestrator can deploy pods of the applications.

Finally, "Quadratic" because it is necessary to consider the weights of communication between pairs of components.

This problem is known to be NP-complete; different solutions have been proposed in literature to solve it through relaxed heuristics such as [128, 129] for quadratic-KP, [130] for multi-dimensional-KP and [131] for multiple quadratic-KP.

The following IQP model does not represent a totally fair solution, from the point of view of the applications, but aims to meet the objectives of Cloud providers: e.g., minimizing energy and inter-rack bandwidth consumption, etc.

Let us denote:

- X^A the number of pods for the generic application A;
- $M = |\mathcal{M}|$ the number of available workers;
- L the number of resource types taken into account (e.g. RAM, CPU, ...);

- $w^A_{j,k}$ the weight of communication between pods j and k, the generic element of the adjacency matrix w_A of the application A;
- $r^A_{j,l}$ the amount of the resource type l required by the pods j of the application A, the generic element of matrix r_A;
- $c_{m,l}$ the amount of available resource of type l in the worker m;
- $\kappa_{m,n}$ the closeness between workers m and n as in eq. (4.8).

The allocation issues of section 4.1 can be resumed in the following multiple quadratic multi-dimensional KP:

$$\max \sum_{A=1}^{N} \sum_{m=1}^{M} \sum_{n=1}^{M} \kappa_{m,n} \cdot \sum_{j=1}^{X^A} \sum_{k=1}^{X^A} x^A_{j,m} \cdot x^A_{k,n} \cdot w^A_{j,k} \tag{4.11}$$

subject to:

$$\sum_{m=1}^{M} x^A_{j,m} = 1 \qquad \begin{array}{l} A = 1 \dots N \\ j = 1 \dots X^A \end{array} \tag{4.12}$$

$$\sum_{A=1}^{N} \sum_{j=1}^{X^A} r^A_{j,l} \cdot x^A_{j,m} \leq c_{m,l} \qquad \begin{array}{l} m = 1 \dots N \\ l = 1 \dots L \end{array} \tag{4.13}$$

Equation (4.11) represents the objective function of the problem: the maximization of the overall flow index of the Cloud *system*.

$x^A_{j,m}$ represents the binary decision variable: it is 1 if the pod j of the application A is placed on the worker m, otherwise it is 0. Constraints (4.12) guarantee that any pod of each application is deployed exactly once; Constraints (4.13) assure that the requirements of a group of pods deployed on the same machine do not exceed the available resources on the specified machine.

Following seminal works [127, 132–134], an auxiliary variable $z_{m,n}^{A,j,k}$ can be exploited to get a linearized version of the presented IQP ((4.11)-(4.13)):

$$\max \sum_{A=1}^{N} \sum_{m=1}^{M} \sum_{n=1}^{M} \kappa_{m,n} \cdot \sum_{j=1}^{X^A} \sum_{k=1}^{X^A} z_{m,n}^{A,j,k} \cdot w_{j,k}^{A} \tag{4.14}$$

subject to:

(4.12) – (4.13)

$$z_{m,n}^{A,j,k} \leq x_{j,m}^{A} \qquad \begin{array}{l} A = 1 \dots N \\ j,k = 1 \dots X^A \\ m,n = 1 \dots M \end{array} \tag{4.15}$$

$$z_{m,n}^{A,j,k} \leq x_{k,n}^{A} \qquad \begin{array}{l} A = 1 \dots N \\ j,k = 1 \dots X^A \\ m,n = 1 \dots M \end{array} \tag{4.16}$$

$$x_{j,m}^{A} + x_{k,n}^{A} \leq 1 + z_{m,n}^{A,j,k} \qquad \begin{array}{l} A = 1 \dots N \\ j,k = 1 \dots X^A \\ m,n = 1 \dots M \end{array} \tag{4.17}$$

$$z_{m,n}^{A,j,k} \geq 0 \qquad \begin{array}{l} A = 1 \dots N \\ j,k = 1 \dots X^A \\ m,n = 1 \dots M \end{array} \tag{4.18}$$

Constraints (4.15) assure that, given a pod j for the application A, any $z_{m,n}^{A,j,k}$ can be less or equal to 1 only when m is the worker hosting j. The same happens on constraints (4.16) just for the pod k if hosted on worker n. Constraints (4.18) assure that $z_{m,n}^{A,j,k}$ are non-negative values.

Thus, if the pod j and the pod k are not hosted, respectively, on the workers m and n, the auxiliary variable is constrained to be 0; otherwise, it belongs to $]0, 1]$. Finally, constraints (4.17) guarantee that the auxiliary variable can only lay in $\{0, 1\}$. Therefore, $z_{m,n}^{A,j,k}$ is 1 only if, given the application A, pod j is hosted on worker m and pod j is hosted on worker n.

The IQP (4.11)-(4.13) has been reduced into a Mixed Integer Programming model (MIP).

4.4 A case study: impact of closeness and isolation on the performance of a 3-tier application

As discussed in the previous sections, the performance of a distributed application strongly depends on its underlying network's performance, especially when the communication between its microservices is intensive. A straightforward example is given in Table 4.2 that shows the result of the same test, namely a single-value read in a database, relevant to 3 different real-world scenarios. Column *Network* indicates the selected scenario; The column *Range of values* gives the execution time difference between the longest and the shortest read over 1000 runs.

The table shows a wide range of possible read times, spanning from a low-latency, stable LAN (local AMZ, i.e. Amazon) to a markedly unreliable WAN. In the stable LAN the range of values for the execution times among all the reads is less than 0.2 ms (approximately, the read times are uniformly distributed between 0.435 ms and 0.634 ms) and the standard deviation is 0.046 ms. The unreliable WAN, however, is characterised by high fluctuations where the execution time of the slowest

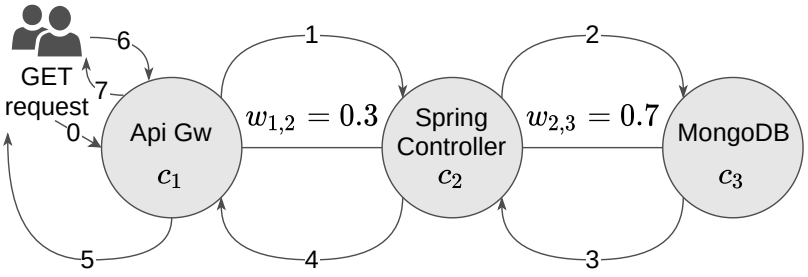

Figure 4.3: Three tier application graph

read (91.02 ms) can be about 180 times greater than the fastest one (0.5 ms).

Table 4.2: Network fluctuation

Network	Range of values	Std.
Local AMZ	0.199ms	0.046ms
Campus MAN	67.430ms	361.959ms
Campus/WAN/AMZ	90.557ms	300.321ms

The experimental results discussed in the following sections will demonstrate how the performance of a distributed application is related to the way its micro-services are deployed and not only to the low level structural parameters of the network, such as throughput, bandwidth, latency, error rate and jitter, of the surrounding network. In particular, it will be shown how the performance of a real test-bed for cloud-native applications depends on the *isolation index* and *closeness* among its micro-services.

4.4.1 Application, workload and performance indexes

The test-bed is based on a simple 3-tier application consisting of a Video API platform on which users can either publish their videos or retrieve

them to be viewed through a generic web video-client. The application is modelled as shown in Figure 4.3 and consists of (i) an nginx API-Gateway (c_1, presentation tier) which acts as the first receiver of an end-user request, (ii) a Java/Spring controller, (iii) a MongoDB database, and (iv) 2 persistent volumes hosted in a GlusterFS storage cluster to provide database files and video files.

The path of the request is also depicted in Fig. 4.3: a GET request reaches the API Gateway (0) and is forwarded to the Controller (1) that retrieve, with 2 queries to the MongoDB database (2, 3), the information about the requested video. Then, the controller replies (4, 5) to the user with a HTTP 302 Status and a location header to redirect itself to the actual video URI. This final video URI reaches the API Gateway (6) which returns the file saved in the *videofiles* persistent volume acting as a web-server (7).

By taking into consideration that the client (i.e., our load test-set) is able to follow HTTP redirects, the measurement at step (7) is limited to the first byte of the videofile.

The application is deployed on an ad-hoc OpenShift cluster consisting of 8 workers hosted at the University of Catania. Each worker is equipped with a virtualized 8-core CPU and 32GB of memory. Links between physical machines hosting the workers consist of a 1Gbps dedicated LAN.

A customized version of *Hey* [135] was utilized as a load generator running on a dedicated host, external to the Openshift Cluster which hosts the application, but within the same network. Deployment of different environment configurations has been executed by exploiting *Ansible playbooks*.

This allows for the measurement of the request execution time (called hereinafter TDT, transaction delivery time) thus avoiding fluctuations

typical of a WAN or similar network infrastructure configuration, as depicted in Table 4.2.

Each execution of *Hey* runs for $600s$, simulating a load of 100 concurrent requests without leveraging Keep-Alive connections.

Each test evaluates the average TDT and its standard deviation. The average TDT is considered as a generic index of the *network speed*: the lower the value, the better the performance. The standard deviation can be instead used in the assessment of the predictability of network performance: the lower its value, the lower the performance variability, a key aspect to take into account when satisfying an SLA.

Moreover, for each test set carried out, the TDT probability mass function is graphed out, fig. 4.4, 4.5.

In the following, 4 different deployment configurations are considered for the components c_1, c_2, c_3:

- case A: the three components are hosted on different workers
- case B: the Spring Controller (c_2) and the API Gateway (c_1) are hosted on the same worker, while MongoDB (c_3) is hosted on a different one;
- case C: the API Gateway (c_1) is hosted on a machine, while MongoDB (c_3) and the Spring Controller (c_2) are clustered together on a different worker;
- case D: each component is hosted on the same worker.

Table 4.3 shows the values of the normalized isolation index according to equation (4.6). Closeness is constant if two micro-services are hosted on different workers, because they are hosted on a unique virtual LAN of the same cluster; thus, for the sake of simplicity, the closeness can be elided.

Table 4.3: Deployment configurations with their normalized isolation index

Deployment configuration	nii_A
Case A	0
Case B	0.3
Case C	0.7
Case D	1

These deployment configurations will be investigated simulating two different scenarios: (i) the "private environment" where the application runs alone, and (ii) the "shared environment" where other applications coexist with it. The performance evaluation of these two different scenarios 1s going to be discussed in the next two subsections.

4.4.2 Performance in the "private environment"

Figure 4.4 shows, for each deployment configuration, the average value, the standard deviation and the histogram of the probability mass function of TDT.

The average and the standard deviation are also summarized in Table 4.4, which compares cases A, B, C vs. the optimal mapping of case D.

Generally speaking, it is clear that as the isolation index and closeness decrease, the average execution time of the transaction increases while also exhibiting a higher standard deviation. The underlying key principle of the proposed approach is that "architectural quality" boosts performance.

Case A has the lowest isolation index and closeness, where the average and the standard deviation of TDT show a nearly five-fold (456%) and two-fold (96%) increase, respectively.

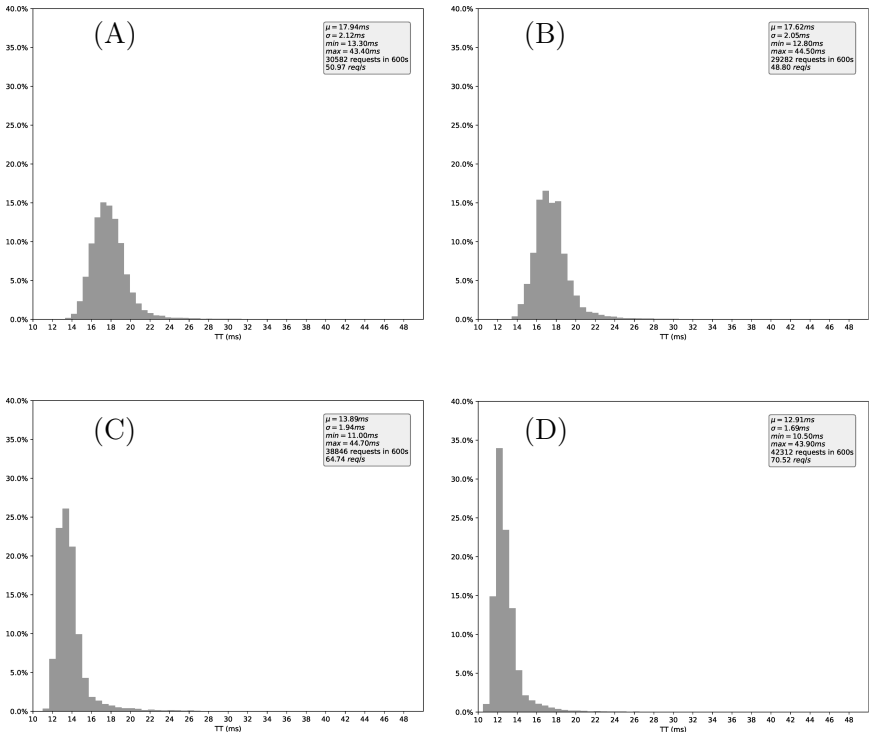

Figure 4.4: TDTs distribution in different deployment cases ("private" environment)

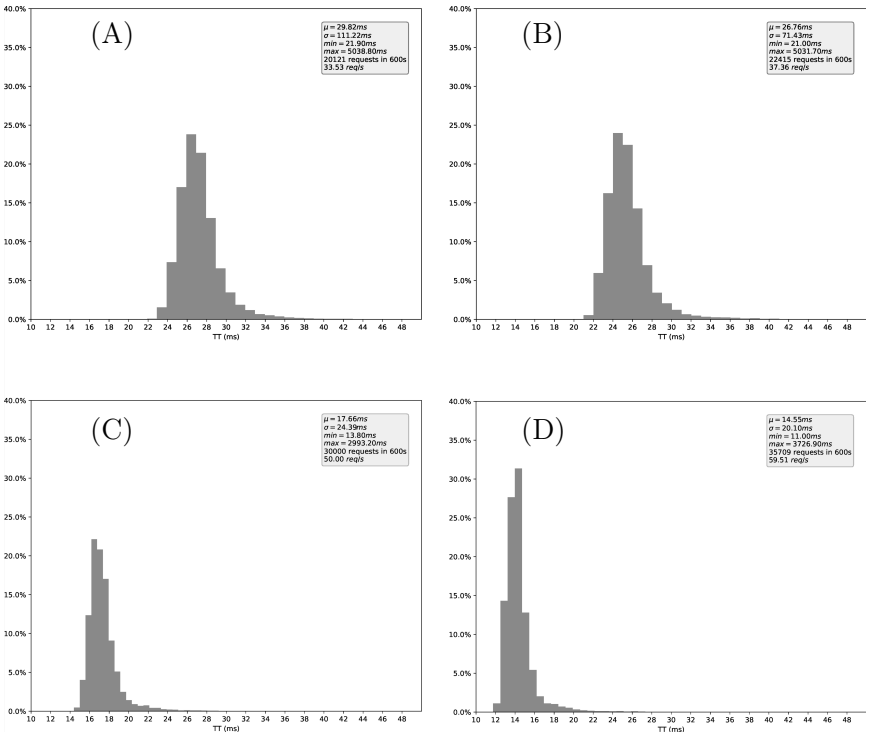

Figure 4.5: TDTs distribution in different deployment cases ("shared" environment)

Table 4.4: Performance of the "private" environment

Configuration	avg	std
Case D (optimal)	12.91ms (100.00%)	1.69ms (100%)
C vs. D (% incr.)	13.89ms (107.59%)	1.94ms (114.79%)
B vs. D (% incr.)	17.62ms (136.48%)	2.05ms (121.30%)
A vs. D (% incr.)	17.94ms (138.96%)	2.12ms (125.44%)

Table 4.5: Performance of the "shared" environment

Configuration	avg	std
Case D (optimal)	14.55ms (100.00%)	20.10ms (100%)
C vs. D (% incr.)	17.66ms (121.37%)	24.39ms (121.34%)
B vs. D (% incr.)	26.76ms (183.92%)	71.43ms (355.37%)
A vs. D (% incr.)	29.82ms (204.95%)	111.22 (553.33%)

Cases B and C show how placing the more coupled components together results in an performance increase of 126% compared to when less coupled micro-services are placed together (as with the database and the API gateway).

These experiments confirm the advantages of inter-worker isolation and the placement of highly coupled pods close together - as confirmed by the fact that, as these parameters worsen, the average TDT times increase.

Furthermore, standard deviation increases too, meaning a larger fluctuation around the average performance is observed, with obvious, adverse, implications affecting QoS/SLAs.

4.4.3 Performance in the "shared environment"

In the second experiment, the network of workers is stressed in order to simulate other communication-intensive applications running concur-

rently with the monitored one. Stressing traffic was injected into the communication links through D-ITG [136]: it is configured as a Daemon-Set in the worker nodes so that each worker intensively communicates with each other. The inter-departure time between packets is taken from a Poisson distribution with mean $100ms$ while the packet size is constant and set to $10KB$.

Figure 4.5 depicts the resulting probability mass function for TDT in any of the deployment configurations.

Average TDT and standard deviation, for deployment configurations A, B, C, are shown, relative to D, in Table 4.5.

Considerations alike to the "private" scenario of Section 4.4.2 apply also to this "shared" scenario: as the isolation index (or FlowIndex) degrades, from case D to A, so do performance metrics. Moreover, with respect to the "private" scenario, the presence of competing applications seems to adversely impact performance further in all cases A, B, C (vs. D), and increasingly so, as architectural quality (isolation and closeness) worsens from case D to A. This is easily observed in Table 4.5 which shows, as already done for the "private" environment, the TDT average and standard deviation compared to the best (shared) case.

On the other hand, the reference application make use of the overlay network links depending on the deployment configuration

When application links are shared with the ones of the D-ITG "simulated" applications, communication interference arises and performance are adversely affected. In case D (isolation index 1, see Table 4.3) the micro-services are on the same worker and do not use links between workers; accordingly, the slight performance degradation observed in Table 4.5 is only due to the additional computing load given by the network stack kernel jobs of the underlying operative system due to the simulated applications running concurrently in the cluster.

In case C (isolation index 0.7, see Table 4.3), the API Gateway is hosted on a worker while the database and the controller are grouped together in another one. The two coupled pods (database and controller) leverage the use of internal communication rather than the shared one.

Communication between the API Gateway and the Controller, though certainly affected by the interference between the applications, is relatively rare compared to the one between the controller and the database; accordingly, both shared and private performance degradation increases slightly: 121.37% and 107.59%, respectively.

Degradation increases in cases B, where the highly coupled pods compete for the shared links with the other applications, and in case A where all the pods communicate over shared links. These two cases show the most measurable degradation: the average values increase by 183.92% in case B and 204.95% in case A.

It is interesting to note what happened for both case C and case B, where the presence of other applications have influenced the average value, increasing it of 121.37% and 183.92% respectively.

However, the normalized increase of standard deviation is almost the same for case C in both "private and "shared" scenarios (114.79% *vs* 121.34%), while for case B there is an important increment of 355.37% which increase till to 553.33% for case A. This increment highlights how strongly the flow index affects the communication quality in terms of fluctuation and, as a consequence, how strongly it affect the performance of a Cloud-native application, both in the case of strict QoS requirements and in a "Best-effort" scenario.

This confirms that applying a strategy which takes into account the indices introduced in this paper, can greatly improve the overall QoS of the hosted services.

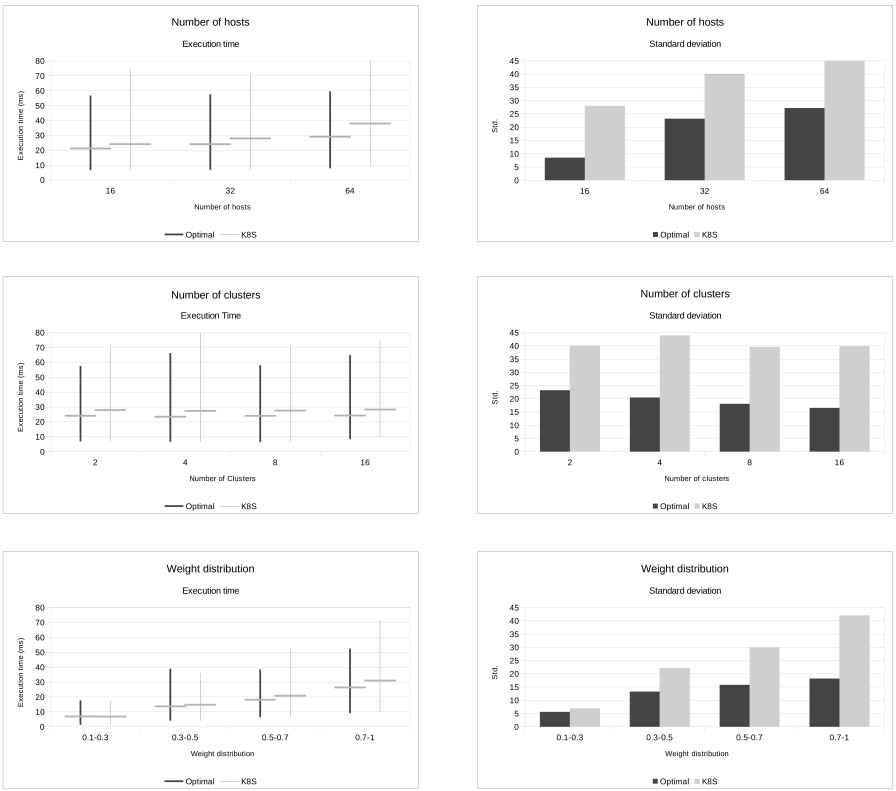

Figure 4.6: Numerical results from simulation environments

4.5 Performance evaluation

The aim of this section is to evaluate, through simulation, how a deployment based on the maximization of isolation and closeness parameters improves the performance of cloud native applications. In particular, the strategy proposed in section 6, based on its IQP formulation using IBM Cplex, is compared with the standard scheduling strategy of Kubernetes.

The Kubernetes strategy considers a queue of pods to place in the cluster. Kubernetes schedules the pods in the queue through the pattern *filter*, *sort*, and *prioritize* [64].

For each pod:

1. a filter selects from the available workers, those that are able to run this pod;
2. a priority-based algorithm associates to any worker a weighted average of scores according to different *score plugins*;
3. the scheduler sorts the set of workers by their score and assigns the pod to the first worker of the ordered queue.

Two of the official Kubernetes *score plugins* [137] utilized in the simulations are the following:

- Least Allocated: it compares the required resources of the pod with the available resources in the worker giving a greater score to the worker with the greatest $available/requested$ ratio;
- Balanced Allocation: favors nodes with balanced resource usage rate. For each node and resource type, it calculates the variance of the $requested/available$ ratio;

The simulation software consist of a graph generator to produce different pods placement scenarios and two handlers which implement (i) a Kubernetes scheduler and, (ii) the optimal IQP as in section 4.3 exploiting the CPlex API.

4.5.1 Generation of applications and cluster graphs

Generation of the graphs for both the applications and workers is randomized.

Table 4.6: Default values for simulations

Number of workers	32
Number of applications	32
Number of workers clusters	2
Containers per application	10
Weight distribution (inter-pod communication)	0.5-1.0
Link	0.8
Max path length	5
Available RAM per worker	32GB
Available CPUs per worker	16

For the campaign of simulations the graphs, both of the applications and of workers are generated through the Erdős-Rényi method [138].

Table 4.6 lists the default values that the generator uses to configure the Cloud system.

The overlay network of workers available for the placement consists of a fully-connected graph of homogeneous workers belonging to the clusters of the cloud system (two by default). As previously described, the applications to be orchestrated consist of a set of weighted graphs, where the nodes are the pods and edges are the links between them, weighted by the communication workload. The pods are generated with a given random probability while the weights of the links got a uniform random distribution.

Finally, the required resources are also randomly generated and are kept consistent with the available resources in the system so that any simulated scenario is actually feasible. In this way, the consistency of the results is guaranteed when varying the input parameters of the simulations.

Three campaigns of simulations are reported in the following:

- varying the number of workers, while keeping constant the total amount of available resources;
- varying the number of clusters;
- varying distribution ranges of inter-pods communication.

For any application placement, the average and the standard deviation of the execution time for each generated request are calculated. These requests are generated as random, self-avoiding walks in the application's graph with a maximum number of steps equal to 5 (a reasonable communication path length for micro-services architecture, as for instance, with different MVC-based components that have a broker for asynchronous messaging).

The execution time of every request is given by the sum of times needed to to traverse each link from a node to the next one in the random walk modeling the request. Execution times are generated following the distributions revealed by the real test-bed in sec. 4.4.

The specific distribution taken into account to calculate the stochastic execution time is chosen by looking at the final status of the links, which are: *internal*, if the two vertices-pods are hosted in the same worker; *private*, if the link is spread across two workers, and is used only by that specific pod's pair; finally, *shared* if the link is spread across two workers and shared with other pods' links.

Each execution time in the sum is weighted by a factor which simulates the closeness between workers: as in Table 4.2, the closeness for workers belonging to the same cluster is a few orders of magnitude higher than that of workers belonging to different clusters.

To reduce the noise in the results, the requests are submitted 10k times.

4.5.2 Numerical results

The simulation results are grouped in Fig. 4.6.

Resource fragmentation

The first two charts report the execution times and their standard deviation while varying the number of workers. The diagrams show how the solution provided by the proposed strategy usually performs better than the Kubernetes one in any case. In the scenarios of 16 and 32 workers, both IQP and Kubernetes are able to deploy most of the applications with a normalized isolation index 1, because the available resources are not too fragmented. Execution times given by IQP, on average, are 13% (16% in the 32 worker scenarios) lower.

Moreover, while the fragmentation of resources becomes higher, the proposed strategy ever performs better than Kubernetes: with 64 workers (resources halved on each worker with respect to the 32 workers scenario), there is an improvement of 30% in the average execution time.

Looking at the standard deviations, the impact of using a network-aware strategy based on the performance indices described in this paper can afford a big improvement of the quality of service given to the applications. The higher the fragmentation of the resources, the higher the standard deviation given by the schedulers, but IQP got a 30% to 60% lower standard deviation than the Kubernetes scheduler, so assuring lower network fluctuations.

Network fragmentation

The second campaign compares the proposed strategy with Kubernetes as the number of clusters varies.

Having a link traversing a different clusters can afford to strong degradation of execution time due to the underlying physical network characteristics.

The average response time measured when using Kubernetes scheduler is greater than the one reported by the IQP solutions: applications deployed with the Kubernetes strategy reported a 15% higher average response time than the IQP. As in the previous simulation the proposed strategy provides a lower lower standard deviation, going from 59% (2 clusters) to 41% (16 clusters) than the Kubernetes one. What emerges from the performance analysis is that a strategy based on the flow index represent strong improvements in environments like Multi-cloud, edge, and fog computing, where the underlying networks cannot be controlled by all the stakeholders (i.e., network operators, application providers, ...) and are fragmented and variable rather than the ones available, for example, in a Spine-Leaf based cloud data-center.

Varying distribution of weight

As said before, the proposed approach fits well in the orchestration of Communication-Intensive applications.

The last campaign of simulation compares the performance of the two schedulers while varying the parameters of the uniform distribution exploited for the communication links weights.

The results confirm how the proposed strategy is mainly useful for communication intensive applications: choosing a weight distribution taken in the range $[0.1, 0.3[$, the performances reported by Kubernetes and IQP are pretty similar, especially in terms of average response time.

Getting over the threshold of the distribution range $[0.5, 0.7[$, the proposed strategy reports a pretty constant standard deviation, while

Kubernetes increases it till to 230% than the network-aware strategy when the weight is chosen in the range $[0.7, 1]$.

This is an important finding of the proposed approach. The lower the standard deviation, the higher the statistical predictability of execution times: the proposed strategy is able to get more predictable response times and, as a consequence, to improve the QoS of applications, especially in the case of Communication-Intensive applications, giving better warranties on SLAs.

4.6 Conclusion

This chapter introduced a set of mapping strategies to improve the QoS management of communication intensive applications in PaaS clouds. Relying on the flexibility provided by the containerization architecture and on the knowledge of the application's structure, a set of best practices aims to make the performance of cloud workflows predictable by minimizing both the mutual interferences between the workflows and the number of costly run-time adaptation procedures.

To do that, the proposed strategies take into account two specific parameters, i.e., the *closeness* among workers hosting components and the *isolation index*, an important parameter introduced for measuring the degree of mutual influence among micro-services.

The effectiveness of the proposed approach has been evaluated by means of a simulation campaign aimed to show how the *isolation index* and *closeness* parameters influence the performance of workflows running in PaaS cloud environments.

In particular, these simulations have demonstrated that the combined use of *isolation index* and *closeness* parameters in mapping strategies allows the cloud provider to reduce the interference among workflows,

to increase the average communication speed-up and, as a consequence, improving both the QoS management and resources utilization.

The next chapter, will focus on a similar approach focused to the field of cloud-edge offloading, and therefore the allocation of partial applications on edge cluster by the use of the service elasticity principle.

Chapter 5

MORA: Multiple Option Resource Allocation on Edge Computing environments

While the approach presented in the previous chapter focuses on the exploitation of service providers' information about the workflow of their applications, the aim here is to focus on how the service providers' can interact with network operators to deploy a part of their services to the Edge. The considerations that will be presented are materialized into a polynomial time algorithm - MORA - and an architecture to enable allocation of containers in the Edge and the offloading of services between edge and cloud.

As anticipated, despite the scenarios seem different, the work of the previous chapter and MORA are strictly related.

In particular, both need interaction between the resource providers (i.e., Cloud providers, or network operators in the following); the strategy of previous chapter can also be applied on smaller networks like Fog

Table 5.1: Summary of the notation.

Parameters	
M	Number of nodes
N	Number of service providers
J^i	Number of options by SP i
$Z^{i,j}$	Number of containers for option j of SP i
$c_{l,m}$	Amount of resource l in node m
$w_{l,z}^{i,j}$	Amount of resource l required by the container z of option j by SP i
$u^{i,j}$	Utility given by choosing option j of SP i

Decision variables	
$x^{i,j}$	Binary variable, 1 if the option j by SP i is chosen, 0 otherwise
$y_{z,m}^{i,j}$	Binary variable, 1 if the container z of option j by SP i runs on node m, 0 otherwise

and Edge computing facilities to improve internal communication performance and, therefore, enhancing the requirements for a very low-latency network at the Edge.

The core of MORA is that (i) it exploits service elasticity, i.e., the fact that services can adapt to the resources allocated by the NO and rely on a remote Cloud for the excess of computation, (ii) it is suitable for micro-services architecture, which decomposes a single service in a set of components, which MORA places in the different computational nodes of the Edge and (iii) it copes with multi-dimensional resources, e.g., memory and CPUs.

Let's consider the case of a Network Operator (NO) owning an Edge Computing infrastructure, composed of $m = 1, \ldots, M$ nodes. These cluster nodes may be servers installed on a Central Office or machines installed in a base station. They are virtualized through a hypervisor and exploit linux kernel capabilities like *cgroups* and *namespaces* in order to separate run environments of the services, in containers, so that third

party Service Providers (SPs) can concurrently run their services there. In short, resource accounting and execution environment can be based on solutions like Kubernetes and OpenShift. Each node has a limited amount of each resource type. Resources are of type $l = 1, \ldots, L$. In the numerical results it will be $L = 2$, with resource types being RAM and CPU. Each node m has a capacity $c_{l,m}$, which is the amount of resource of type l available. N service providers compete ($i = 1, \ldots, N$) to use the resources available at the edge to run their containerized applications. A one-to-one mapping between *service* and *service provider* is assumed as if each service provider can run at most one application on the Edge: actually, this doesn't lack of generality, since a SP could declare itself multiple-time as multiple service providers. Similar to [139], in this work it is considered that there is no unique way to run a service at the edge. Each service is decomposed in a set of micro-services, each hosted in a container. Moreover, each service i can run in multiple possible configuration options $j = 1, \ldots, J^i$. Each configuration option j of service i requires concurrently running a set of containers $z = 1, \ldots, Z^{i,j}$. Each SP i declares the possible configurations under which it is capable to run and the NO decides (i) which configuration option to accept and (ii) for all the containers belonging to that option, which node they should run to.

These decisions are based on utility and resource consumption.

Ideally, the NO would like to choose for each SP the option that provides the largest utility. Unfortunately, this is in general not possible, due to the scarcity of resources available in the Edge nodes. Indeed, each container consumes resources. Each Edge node can host different containers, from different SPs. Obviously, the sum of resource type l consumed by the containers running in a certain node m cannot exceed its capacity $c_{l,m}$, for any resource type $l = 1, \ldots, L$. Therefore, the NO

must optimally choose one option per SP, or a *null* one, i.e., choosing to not serve that SP. The NO must resolve a trade off between utility and resource consumption; at the same time, it has to optimally place the containers of the chosen options in the available Edge nodes.

If abundant resources are available, a service can be configured in order to exploit them all, thus almost completely running at the edge. If less resources are available, the service may configure itself so to adapt to those and to move some of the computation and data to remote servers or cloud computing infrastructures.

The multiple configurations in which a Service Provider (SP) can run its service at the edge denotes its capability to adapt to different amounts of available resources.

Each configuration results in a certain utility $u^{i,z}$ for the Network Operator (NO), that in the simplest case represents bandwidth saving [57], which is the objective considered for the numerical results. Utility can in general be cost savings [77], QoS or fairness [58], elaboration time savings [56], depending on the application and the information available. As commonly done in the literature [58, 77, 140], this work is based on the assumption that the resources needed for the configurations and the other characteristics of the configurations are known at the moment of taking the resource allocation decision. This can easily be achieved, by the use of frameworks as Tosca [141] or OpenSLO [23], in line with today containerized environments. For example, in Kubernates it is possible to define memory, CPU and bandwidth limits when Deployment files are submitted.

Note that the utility is defined per-option, not per-container. The rationale is that running containers individually is not useful at all. For instance, to provide an on-line gaming service, we might need a container for authenticating users, another for retrieving video frames and another

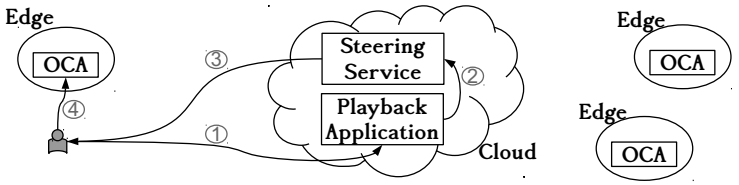

Figure 5.1: Netflix architecture for OCAs CDN

to transcode them. They might all be needed together. Running the authentication server alone, may be senseless. Either the containers of an option run all or no one. Therefore, utility comes from the concurrent run of all the containers of a configuration option, and not from any single container. Since the different sets of containers that can collectively provide a service depend on the service itself, it is in the business of the SP to declare the possible configuration options.

5.1 Architecture

This section will present an architecture that can enable what is being proposes. In order to start depicting the architecture into an existing and practical technology, a briefly outline of successful solution widely adopted by Netflix is hereinafter described.

5.1.1 An existing implementation of Edge Computing

Netflix is one of the largest content providers. It deploys its own hardware appliances, called Open Connect Appliances (OCAs) [142], into Internet access networks. OCAs store a part of the content catalog and can serve directly a fraction of local users' requests, without generating inter-domain traffic. For this reason, NOs often accept to install this hardware in their premises. Requests are processed as in Fig. 5.1: (1) A

user requests a video. (2) A micro-service in the Cloud selects the files to be sent to the user. (3) The steering service determines the OCAs closest to the user based on its IP address and generates a list of URLs pointing to the OCAs. (4) The user client uses the URLs list to play the video.

This generates a utility to the NO, in terms of inter-domain traffic saving. The limit of this solution is its limited *permeability*: it is unfeasible in terms of cost and physical space to install hardware appliances to the very edge of the network, i.e., in many base stations, central offices, access points, etc. Moreover, it is infeasible that hundreds of SPs, like Youtube, Netflix, gaming providers, IoT providers, etc., will each install physical boxes into thousands of access networks: installing and maintaining such physical infrastructure would have an enormous cost for both SPs and NOs. Moreover, there is no physical space to host many physical boxes in the network locations at the Edge. However, the case of the OCA shows that both SPs and NOs have interest in EC, to run services at the Edge. These limits can be overcome if appliances are virtualized, as it is already done in Cloud environments. To make EC feasible, this work proposes to let the NO owns the computational resources and to virtualize them, in order to allocate them to third party SPs, acting as tenants. Each SP can then behave individually similarly to Fig. 5.1. The NO allocates slices of resources to several third party SPs. The SP can then use its assigned slice as it were a dedicated hardware. Memory encryption technologies [143] can guarantee that data and processing remain inaccessible to the NO, even if they run in its premises. Note that, while big players may continue to use their hardware appliances, a virtualized solution owned by the NO is probably the only way small or medium SPs can reach the Edge of the network.

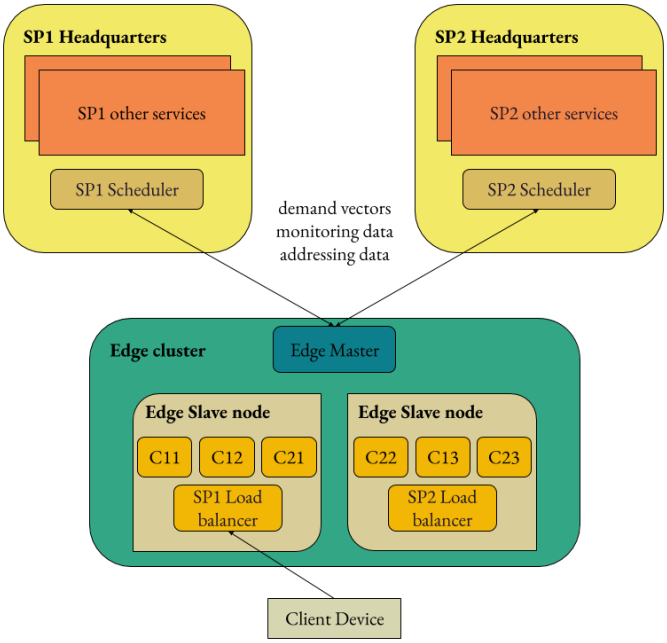

Figure 5.2: Overview of the proposed architecture. SPs run part of their service in their premises or in remote Clouds, which we denote as Headquarters.

5.1.2 Proposed architecture

The components of the proposed architecture are (Fig. 5.2):

- *Edge slave nodes*: owned by the NO, they run the SPs' containers.
- *Edge Master*: a process controlled by the NO, responsible for (i) monitoring resource usage (e.g. using fine grained monitoring functions available in containerized environments like Kubernetes [144]); (ii) collecting the different deployment options from SPs; (iii) deciding the options to be deployed; (iv) informing the

SPs about the authorized options and receiving back the containers descriptors (e.g. Dockerfile or Pods YAML); (v) running the containers in the Edge slaves. The optimization strategy of § 5.2 runs in the edge master.

- *SP Scheduler*: each SP has its own scheduler; First, it declares the set of possible configuration options to the Edge Master, specifying resource requirements and utility. After the Edge Master selects one of these options, the SP Scheduler forwards to the Edge Master the relative containers descriptor files to deploy its application at the Edge;

- *SP Load balancer*: each SP has its own load balancer; it intercepts user requests as in [145] and, based on the amount of requests served by the Edge it decides to forward the request to a remote Cloud or to handle it within the Edge [71].

5.1.3 Edge Master workflow

The Edge Master is the core component of the proposed architecture. Periodically, it performs the following operations.

1. It monitors the available resources and receives the set of options from the SPs schedulers; it is given as a list containing, for each option, the relevant amount of utility estimated and information on the resource requirements for each container;

2. It executes the placement algorithm to select the best option for each SP according to the collected information in point (1). The decision is sent to the SPs schedulers;

3. It receives the (chosen) option descriptors files (e.g. Dockerfiles, Smarm configurations files, Kubernetes YAML...) for the authorized options and runs these containers in the slaves nodes;

4. Finally, it communicates to SPs' load balancers the addressing data to reach the Edge internal containers. Based on the occupied resources the load balancers redirect the user requests to the Edge resources or to the Cloud.

5.2 Optimal resource allocation

The NO aims to maximize its overall utility, i.e., the sum of the utilities coming from all the selected options. In order to do so, the NO must concurrently take two decisions

- *Option selection*: the NO must select at most one configuration option per SP.

- *Container placement*: the NO must deploy each container of the selected options to one of the available nodes

The following is an Integer Linear Programming (ILP) formulation of the problem. The decision variables modeling the Option selection are $x^{i,j}$, which is 1 if the j-th option of the SP i is chosen. Container placement is instead represented by the decision variables $y^{i,j}_{z,m}$, which is 1 if the z-th container of the j-th option of SP i is placed on node m.

The mathematical notation is summarised in table 5.1.

$$\max \sum_{i=1}^{N} \sum_{j=1}^{N_i} u^{i,j} \cdot x^{i,j} \tag{5.1}$$

$$s.t., \sum_{m=1}^{M} y_{z,m}^{i,j} = x^{i,j} \qquad \begin{array}{l} i = 1 \dots N \\ j = 1 \dots J^i \\ z = 1 \dots Z^{i,j} \end{array} \tag{5.2}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{J^i} \sum_{z=1}^{Z^{i,j}} y_{z,m}^{i,j} \cdot w_{l,z}^{i,j} \leq c_{l,m} \qquad \begin{array}{l} l = 1 \dots 2 \\ m = 1 \dots M \end{array} \tag{5.3}$$

$$\sum_{j=1}^{J^i} x^{i,j} \leq 1 \qquad i = 1 \dots N \tag{5.4}$$

$$x^{i,j}, y_{z,m}^{i,j} \in \{0,1\} \qquad \begin{array}{l} i = 1 \dots N \\ j = 1 \dots J^i \\ z = 1 \dots Z^{i,j} \\ m = 1 \dots M \end{array} \tag{5.5}$$

The objective is to maximize the utility (5.1), setting the binary variables $x^{i,j}$. Equations (5.2) guarantee that each container z of the chosen option j by the SP i ($x^{i,j} = 1$) is deployed ($\exists m \in \{1 \dots M\} : y_{z,m}^{i,j} = 1$). Constraints (5.3) guarantee that the sum of the requirements for the set of containers deployed on a node m for each resource l is less than the total amount of available resources in node m so that these containers can actually run on the node. Finally, the constraints (5.4) guarantee that a service provider can deploy at most one option in the Edge cluster.

If we have one only option per SP and a unique dimension, e.g. memory, the problem is similar to a Set-union Knapsack problem [146] and it has been solved via Dynamic Programming or via bio-inspired algorithms like bee-colony optimization [147]. If we have a single node, we can just consider, for each option, the total memory and the total CPU

needed by all the containers composing the option. We can thus forget about the different containers and in this case we have a Multiple-Choice Multi-Dimensional Knapsack Problem (MCMDKP) [148], like in [78], although the authors do not clearly state it. Considering just one option per SP and one node, the problem reduces to a multi-dimensional knapsack problem (l-KP), which is a challenging problem. Methods based on the Lagrangian dual exist but difficult to apply in practice (Sec.9.2 of [149]). Moreover, Fully Polynomial Time Approximation Schemes cannot exist unless P=NP (Sec.9.4.1 of [149]), which motivates the several greedy-type heuristics proposed in the literature (Sec.9.5 of [149]). However, they cannot be directly used in our problem, which is more complicated than l-KP, since we need to cope with multiple options, nodes and containers.

Proposition 1. *Problem P is NP-hard.*

Proof. As already anticipated, P reduces to a Knapsack Problem with $L = 1, M = 1, J^i = 1, i = 1, \ldots, N$ and $Z^{i,j} = 1, i = 1, \ldots, N; j = 1, \ldots, J^i$, which is NP-hard. □

The problem is complex, and there is no possibility to construct Fully Polynomial Time Approximation Schemes, as §9.4.1 of [149] shows that they cannot exist (unless P=NP), already for the simpler case of $M = 1, Z^{i,j} = 1$ and $J^i = 1, i = 1, \ldots, N$, which is known as l-KP.

5.3 MORA

The MORA heuristic uses aggregate values for the resource requirements and availability, in order to neglect, at a first stage, the complexity represented by the fact that resources available are scattered across different

nodes, resource required are split in different container requirements and requirements are multi-dimensional.

To this aim, the overall resource requirements of an option j of a SP i is defined as

$$w_l^{i,j} = \sum_{z=1}^{Z^{i,j}} w_{l,z}^{i,j}. \tag{5.6}$$

A parameter $h_l \geq 0$, called "relevance value", is introduced with a role similar to the relevance values in §9.5.1 of [149].

Finally, the generalized resource utilization of an option j of a SP i is being defined as:

$$w^{i,j} = \sum_{l=1}^{L} h_l \cdot w_l^{i,j} \tag{5.7}$$

To ease computation, MORA heuristic algorithm does not consider all the possible options, but it first removes the *dominated options* and then *LP-dominated* options, defined as follows, which do not provide significant utility gain with respect to the resources they require.

The following definitions are needed to deep dive into the concepts of options dominance, efficiency and options jump, i.e., comparing the utility of options when choosing the ones that maximize utility. Finally, LP-dominance is defined for the scenario in this work.

Definition 4. *For any SP i, an option j is dominated by another option $j' \neq j$ iff (i) $u^{i,j'} > u^{i,j}$ and $w^{i,j'} \leq w^{i,j}$ or (ii) $u^{i,j'} \geq u^{i,j}$ and $w^{i,j'} < w^{i,j}$. An option is dominated, if it is dominated by some other option.*

Definition 5. *For a service provider $i = 1, \ldots, N$, the efficiency of a jump $j \to j'$, where $w^{i,j'} > w^{i,j}$ is:*

$$e^{i,j \to j'} = \frac{u^{i,j'} - u^{i,j}}{w^{i,j'} - w^{i,j}} \tag{5.8}$$

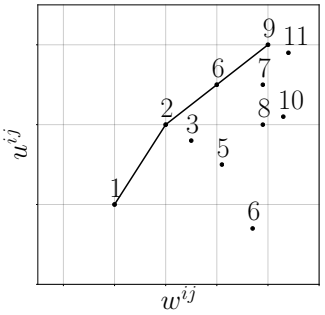

Figure 5.3: Example of set of options of a SP i. The options connected by the line constitute the ordered list of LP-extremes.

Definition 6. *A non dominated option j of a SP i is LP-dominated, if there exist other non dominated options j', j'' such that $u^{i,j'} < u^{i,j} < u^{i,j''}$, $w^{i,j'} < w^{i,j} < w^{i,j''}$ and $e^{i,j'\to j''} \geq e^{i,j'\to j}$ The option j is an LP-extreme if it is neither dominated nor LP-dominated.*

The names "LP-dominance" and "LP-extremes" come from the fact that the concept is related to the LP-relaxation of the Mutliple Choice Knaspack Problem, but this is not relevant for the scope of this proposal. Fig. 5.3 illustrates the concept of LP-extremes, similarly to Fig. 11.1 of [149]. Now, sorting the LP-extreme options produces the list defined as follows:

Definition 7. *For each SP i, we denote with $\underline{j}^i$ the list of its LP-extreme options. We denote with $\underline{j}^i[k]$ the option at its k-th position. This list is ordered in increasing values of $w^{i,j}$, such that $w^{i,j^i[k]} \leq w^{i,j^i[k+1]}$. In the first position of such list we add a fictitious "null" option, such that $w^{i,j^i[0]} = u^{i,j^i[0]} \triangleq 0$.*

Proposition 2. *For any SP i, the list $\underline{j}^i$ of LP-extremes can be computed in $O(|J^i| \cdot R^i)$, where R^i is the number of LP-extremes.*

Proof. Ch. 11 of [149] shows that LP-extremes correspond to the convex hull of the set of options $j = 1, \ldots, J^i$. To compute the convex hull we use [150], which has the complexity above. □

5.3.1 MORA Algorithm

The Multiple Option Resource Allocation (MORA) algorithm is shown in Algorithm 5.1. It takes as input the set of parameters of the ILP (5.9)-(5.4) describing the scenario plus a configuration parameter h_l for $l = 1, \ldots, L$. The algorithm returns a solution, i.e. values for any decision variable. The algorithm solves two decision problems that the NO must solve: (i) *Option selection*: which option (or configuration) per SP must be accepted (variables $y^{i,j}_{z,m}$) and (ii) *Container placement*: in which Edge nodes we should place the containers of the selected options (variables $y^{i,j}_{z,m}$). The pseudo code of Alg. 5.1 is mainly devoted to option selection and calls Alg. 5.2 for the container placement.

Option selection

MORA is iterative. In each iteration, each SP i has a *current position* k^i, which corresponds to the option $j^i[k^i]$. Each SP i has also a *jump efficiency* E^i (line 6), which denotes the efficiency achieved when advancing its position, i.e., the utility gain obtained going from option $j^i[k^i]$ to $j^i[k^i + 1]$ divided by the additional generalized resource utilized. Observe that $u^{i,j^i[k]} < u^{i,j^i[k+1]}$ and $w^{i,j^i[k]} < w^{i,j^i[k+1]}$ by construction, and thus $E^i > 0$.

Then, in each iteration t, the algorithm selects a SP and it checks whether it can change its current option $j^i[k^i]$ to $j^i[k^i + 1]$. Let's say

that service provider i performs a *jump* $j^i[k^i] \rightarrow j^i[k^i + 1]$. As one can expect, the algorithm select the SP whose jump efficiency is the highest (line 12). Let's call this SP the *jumping* SP of iteration t, as it is the one that changes option (the options of the other SPs remain unchanged).

The algorithm, then, tries to place the containers of the jumping SP i^* (line 13). If it succeed, it advances its current option, thus allowing i^* to jump from $\underline{j}^{i^*}[k^{i^*}]$ to $\underline{j}^{i^*}[k^{i^*} + 1]$ (line 15). Otherwise, it removes the option $j^{i^*}[k^{i^*} + 1]$ that cannot be allocated. The algorithm updates the jump efficiency of i^*.

The algorithm terminates when the lists $\underline{j}^i$ of all the SPs have been visited (line 10).

Container placement

The placement operations are described in Alg. 5.2. The algorithm works by constructing a *tentative placement* $\hat{y}^{i,j}_{z,m}, \forall i, j, z, m$. If it is able to construct a feasible tentative placement, i.e., it is able to place all the above-mentioned containers in the available nodes without violating the resource constraints, it updates the actual placement $y^{i,j}_{z,m}$ accordingly (Line 21). Otherwise, it ignores the tentative placement and we leave the actual placement unchanged.

The tentative placement is practically identical to the actual placement (Line 1), except for the containers of the jumping SP i^*. Since the objective is to place the containers of the new option j^* of SP i^*, it first reset all its previously selected options (Line 2). Then, it iterates through the containers of option j^* of SP i^*, and it tries to place them one by one. In order to place a container z, a first check is to select the candidate nodes $\mathcal{M}(z)$ whose residual capacity is enough to host it (Line 7) and the algorithm choses one of them (Line 10).

Similarly to Sec.III.C of [151], this choice is based on the product of residual capacities, but we use arg max while [151] chooses arg min.

To summarize, at each iteration the algorithm takes a hierarchical decision: it first selects an option of a service provider, based on the best jump concept. Then, it tries to place the composing containers in the available nodes. Note that the operations within each iteration does not correspond to any change to the actual resource allocation. The algorithm is always executed until the terminating condition, and only after that the result is taken to decide the actual resource allocation.

5.3.2 Properties of MORA

In this section, properties of MORA in terms of time complexity are discussed together with an upper bound of the problem and a discussion about the impact of the algorithm parameters h_l.

Computational Complexity

Proposition 3. *The time complexity of MORA is* $O(N^2JRZML))$*, where* J *is the maximum amount of options per SP and* Z *is the maximum number of containers per option and* R *the maximum number of LP-extremes per SP.*

Upper bound

Knowing the upper bound of P is important, since we can compare it with the utility provided by the MORA heuristic and verify how far it is from the optimum. Moreover, MORA is anytime, i.e., if one terminates it at any iteration, it returns a valid allocation. The distance from the upper bound can guide in the decision whether to continue the iterations or not, which can potentially save computation time. In order to do so, fix any values for $h_l, l = 1, \ldots, L$, compute $w^{i,j}$ as in (5.7) and $c_{\text{tot}} \triangleq$

Algorithm 5.1 MORA algorithm.

Input: $u^{i,j}, w^{i,j}_{l,z}, c_{m,l}, h_l$.

Output: $x^{i,j}, y^{i,j}_{z,m}$, upper bound $\hat{u}$.

// Initialization

Set $x^{i,j} := y^{i,j}_{l,m} := 0$ for all l, m and all options i, j

for all SP $i := 1, \ldots, N$ **do**

  Compute $w^{i,j}, j = 1, \ldots, J^i$, as in (5.7).

  Compute the ordered list $\underline{j}^i$ of options of SP i as in Def. 7.

  Initialize the current position $k^i := 0$ on such list.

  Compute

$$E^i := \begin{cases} e^{i,\underline{j}^i[k^i] \to \underline{j}^i[k^i+1]} & \text{if } k^i + 1 \neq \text{ end of the list} \\ -\infty & \text{otherwise} \end{cases}$$

end for

// Main loop

for Iteration $t := 0, 1, \ldots$ **do**

  if $E^i = -\infty$ for $i := 1, \ldots, N$ **then**

    break // We arrived at the end of all lists $\underline{j}^i$.

  else

    $i^* := \arg\max_i E^i$ *// Jumping SP*

    success := placeContainers($i^*, \underline{j}^i[k^i] + 1$) *// see Alg. 5.2*

    if success = True **then**

      $k^{i^*} := k^{i^*} + 1$ *// Advance current option*

    else

      Remove the $k^{i^*} + 1$-th element of the list $\underline{j}^{i^*}$.

      // Note that, now the option that was in the

      // $k^{i^*} + 2$-th position (if any), now goes to the

      // $k^{i^*} + 1$-th position.

    end if

    Update

$$E^{i^*} := \begin{cases} e^{i^*,\underline{j}^{i^*}[k^{i^*}] \to \underline{j}^{i^*}[k^{i^*}+1]} & \text{if } k^{i^*} + 1 \neq \text{ end of the list} \\ -\infty & \text{otherwise} \end{cases}$$

  end if

end for

//Translate to ILP notation

Set $x^{i,j} := 1$ for $j = \underline{j}^i[k^i]$ if $k^i > 0$, for any SP i.

return $x^{i,j}, y^{i,j}_{z,m}$.

Algorithm 5.2 Container placement algorithm.

Input: i^*, j^*
Output: boolean *success*.

$\hat{y}^{i,j}_{z,m} := y^{i,j}_{l,m}, \forall z, j, l, m$
// Release the containers of the current option of i^*:
$\hat{y}^{i^*,j}_{z,m} := 0, \forall j, z, m$
// Compute the residual capacity given by the tentative placement:
$\hat{c}_{l,m} := c_{l,m} - \sum_{i=1}^{N} \sum_{j=1}^{J^i} \sum_{z=1}^{Z^{i,j}} \hat{y}^{i,j}_{l,m} \cdot w_{l,z}$
success := True
for all $z := 1, ...Z^{i^*,j^*}$ **do**
  // See which nodes can host container z:
  $\mathcal{M}(z) := \{m \in \{1, \ldots, M\} | w^{i^*,j^*}_{l,z} < \hat{c}_{m,l}, l = 1, \ldots, L\}$
  if $\mathcal{M}(z) \neq \varnothing$ **then**
    // Select one of those nodes:
    $m(z) := \arg\max_{m \in \mathcal{M}(z)} \prod_{l=1}^{L} \hat{c}_{l,m}$
    $\hat{y}^{i^*,j^*}_{z,m(z)} := 1$ // Assign the container to the selected node
    $\hat{c}_{l,m} := \hat{c}_{l,m} - w^{i^*,j^*}_{l,z}$ // Update the residual capacity
  else
    // It is not possible to place container z,
    // and thus the entire option
    success := False
    break
  end if
end for
if success = True **then**
  // The tentative placement is accepted as actual placement
  $y^{i^*,j^*}_{z,m} := \hat{y}^{i^*,j^*}_{z,m}, \forall z, m$
end if
// Else, we leave the actual placement unchanged
return success

$\sum_{m=1}^{M}\sum_{l=1}^{L} h_l \cdot c_{l,m}$ and resort to a problem known in the literature as Multiple Choice Knapsack Problem (MCKP):

$$\max \sum_{i=1}^{N}\sum_{j=1}^{N_i} u^{i,j} \cdot x^{i,j} \qquad \text{(MCKP)} \quad (5.9)$$

subject to

$$\sum_{i=1}^{N}\sum_{j=1}^{J^i} x^{i,j} \cdot w^{i,j} \leq c_{\text{tot}}; \qquad \sum_{j=1}^{J^i} x^{i,j} \leq 1; \qquad x^{i,j} \in \{0,1\} \qquad \begin{matrix} l = 1 \dots L \\ i = 1 \dots N \\ j = 1 \dots J^i \end{matrix} \quad (5.10)$$

Since a solution that satisfies (5.2)-(5.4) also satisfies (5.10), the optimal solution of MCKP is an upper bound to the optimal solution of P. An upper bound of the original problem P can be obtained by using the algorithm from Dyer and Zemel (Fig. 11.5 of [149]) that computes in linear time the optimal solution of the LP-relaxation of MCKP.

Proposition 4. *An upper bound $\hat{u}$ of the original problem P can be found in $O(\sum_{i=1}^{N} J^i)$.*

Impact of the relevance values

The relevance values $h_l, l = 1, \dots, L$ are algorithm parameters that change the placement results. Indeed, changing h_l, the values of $w^{i,j}$ change for all js (see (5.7)) and thus the list $\underline{j}^i$ changes as well. This value serves to weight resource types among them. If, for example, a certain resource type l, say memory, is scarce in the Edge, the algorithm should tend not to select options that consume a lot of resource l. This can be achieved by setting a high value of h_l. By doing this, an option i, j that consumes a lot of l-resource would have a high $w^{i,j}$, and thus would have less chances to be in the LP-extremes list $\underline{j}^i$ (it would tend to be on the right of Fig. 5.3). Moreover, jumping from another option j' to j would

likely result in a low efficiency $e^{i,j' \to j}$ and Alg. 5.1 would prefer other jumps. Observe also that different values of h_l would result in different upper bounds $\hat{u}$. In this way, one can compute different upper bounds and just consider the minimum value.

5.4 Numerical results

MORA has been simulated to show how enabling service elasticity by allowing multiple configuration options to SPs notably improves the utility of the Edge. Simulations of MORAof have been run to compare the performance of MORA, computed with the ILP (§ 5.2), with a *naive* allocation, which consists in randomly option selection and container placement. The code of the ILP in glpk and the python code to orchestrate the simulation are available as open-source [10], together with the scripts to reproduce the results presented here. The simulations run in a Intel Xeon CPU E5-4610 @ 2.30 GHz with 256GB RAM, the results are averaged across 20 runs and 95% percentiles are reported. Finally, MORA has been simulated by using publicly available traces from Google and Alibaba clusters. While the former simulations allow to study the sensitivity of MORA to selected parameters, real-traces allow to assess performance in real-world cases.

MORA is compared to the *optimal solution* (computed via (5.1)-(5.4) using GLPK) and to a *Naive strategy*. The latter iterates over the available SPs and for each one chooses a random option to be deployed. It then tries to deploy each container of the chosen option in the first node that fits the requirements of the container itself. Whenever the first SP cannot be placed in the Edge cluster the naive algorithm stops. The bad performance that will be shown for Naive demonstrates that is important to select the "right" option per SP and the "right" node per container.

In all plots, all the parameters are kept at their default values (Tab. 5.2) and vary only the parameter(s) explicitly specified. In what follows, the computation time (§5.4.1), the utility achieved and the resources left unused after the allocation (§5.4.1-5.4.1) are evaluated. Other simulation show how resources are distributed among SPs (§5.4.1). In the real traces results, it is being shown the achieved utility varies with number of SPs (§5.4.2). Since MORA is an anytime algorithm, simulations report how the utility evolves during its iterations (§5.4.2).

All results on synthetic scenarios are averaged across 20 runs and 95% confidence intervals are reported, which may not be visible when they are too small. They are calculated on a Intel Xeon CPU E5-4610 v2 @ 2.30GHz with 256GB RAM. The model of the ILP in glpk and the python code of MORA are available as open-source on GitHub [10].

5.4.1 Results on synthetic scenarios

In this scenario, the Edge [70] consists of M identical Intel Xeon nodes with 4 sockets and 4 cores and hyper-threading enabled. Therefore, each node is represented as 16 cores hyper-threaded. $32GB$ RAM are allocated to each of them. For each scenario, N SPs, each declaring the same number J of configuration options. Each configuration option is described in terms of the required Z containers. The memory and the processing required by a container z of the j-th option of service i are drawn from uniform random distributions with mean $\bar{w}_l$, with $l = \{RAM, CPU\}$. They are expressed as dimensionless values for CPUs while the memory is expressed in GB. A fractional value of CPU is to be interpreted as fraction of CPU time. For each scenario, the two values $\bar{w}_{\text{RAM}}$ and $\bar{w}_{\text{CPU}}$ are calculated as follows. First, a *load factor* K is chosen, and then $\bar{w}_l$ computed as

$$\bar{w}_l \cdot Z \cdot N = K \cdot c_{l,\text{tot}}; l = \{\text{CPU,RAM}\} \quad (5.11)$$

where $c_{l,tot} = \sum_{m=1}^{M} c_{l,m}$ is the total amount of resource of type l available at the edge. In other words, on average services request K times the available resources. The default values are reported in Table 5.2.

In all the following plots, only a subset of parameters vary and the others are kept at their default value.

Table 5.2: Default values of the reference scenario evaluated

Number of SPs	N	50
Number of nodes	M	8
Number of options	J	5
Number of containers	Z	8
Load factor	K	1.8

As in [78,139], the utility associated to each option is a random variable in these synthetic scenarios (it will instead be a real value directly taken from the traces in the Alibaba case). Moreover, it is considered that there is a concave relation between the resources used and the utility, which results in a diminishing return. This is a common assumption in the literature [57, 58]: the more resources are used by a SP configuration, the larger one should expect the utility to be, but the additional utility tends to decrease with the resources. Using the notation (5.6) for $w^{i,j}_{\text{CPU}}, w^{i,j}_{\text{RAM}}$, the utility is the following function of the required resources:

$$u^{i,j} = \alpha^{i,j} \cdot \left(\frac{w^{i,j}_{\text{CPU}}}{c_{\text{CPU,tot}}} \right)^{\frac{1}{\beta^{i,j}_{\text{CPU}}}} + (1 - \alpha^{i,j}) \cdot \left(\frac{w^{i,j}_{\text{RAM}}}{c_{\text{RAM,tot}}} \right)^{\frac{1}{\beta^{i,j}_{\text{RAM}}}} \quad (5.12)$$

where $\alpha^{i,j}, \beta^{i,j}_{\text{CPU}}, \beta^{i,j}_{\text{RAM}}$ are sampled, for each option, from random uniform distributions between 0 and 1 for $\alpha^{i,j}$ and between 1 and 5 for $\beta^{i,j}_{\text{CPU}}$ and $\beta^{i,j}_{\text{RAM}}$. Since these parameters are random variable, (5.12) "loosely"

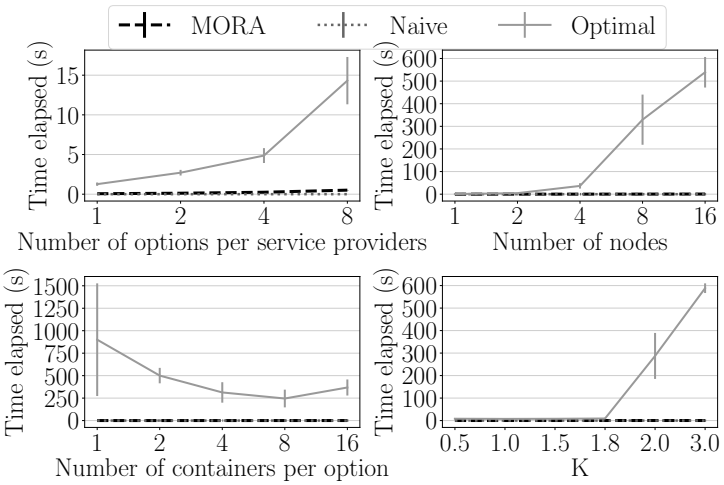

Figure 5.4: Time to compute solutions for ILP, MORA and Naive strategies

show monotonicity and concavity, but is not exactly a monotone and concave function. Infact, (i) in realistic scenarios this relation may not be as "clean" as assuming a perfectly increasing and concave function; (ii) in this way the performance of MORA is tested against pessimistic and 'unclean" situations. Note that this construction follows the assumptions usually adopted in the literature [57, 58, 78, 139]. Real traces will not need these.

Note that, for all feasible options, $u^{i,z} \in [0, 1]$. Since a feasible solution selects at most one option per SP, one can be sure that $u^{\max} := N$ is an upper bound to u^{tot}. The *overall normalized utility* is defined as $u = u^{\text{tot}}/u^{max}$ (5.13). By slight abuse of terminology, in what follows let's refer to "utility" as to indicate the overall normalized utility.

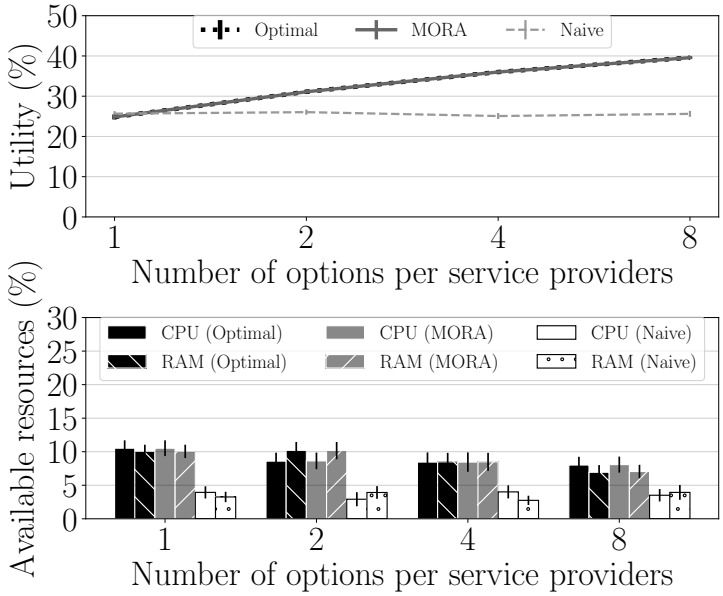

Figure 5.5: Benefits of multiple options.

Time efficiency

Fig. 5.4 shows that the computation of the Optimum from the ILP is too slow for the allocation frequency envisaged in practical deployments, as discussed in chapter 3. On the contrary, MORA remains within 0.05s, as the Naive policy, while also achieving almost optimal utility, as next sections will show..

Benefits of multiple options

Fig. 5.5 shows that utility increases with the number of options per SP. Note that the classic assumption corresponds to the first point of the plot, SP=1. While varying the number of options from 1 to 8 the utility has

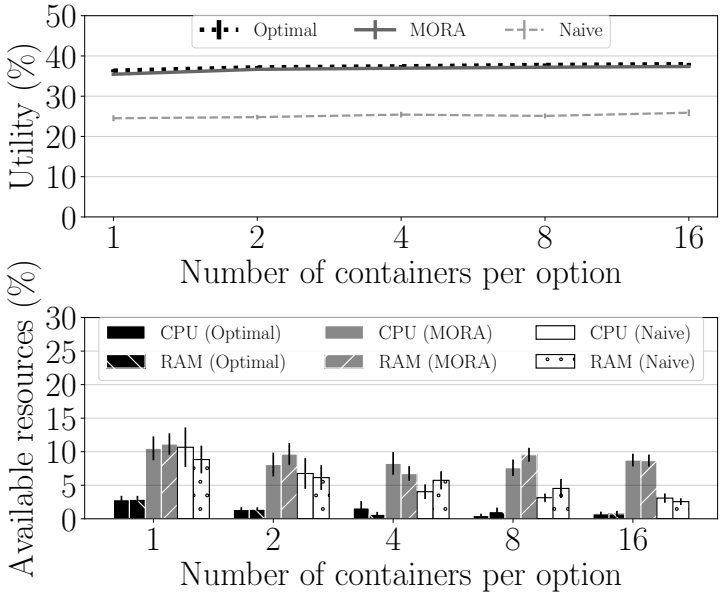

Figure 5.6: Effect of containerization.

a gain almost equal to 60%, which would be lost with classic approaches and which instead we can grasp by exploiting service elasticity. This is equivalent to virtually increase the available resources, by just using them better. Observe also that MORA uses resources as the optimum, while Naive, despite providing poor utility, uses ~3.3 times more resources than optimal/MORA).

The more you containerize, the better the utility

Fig. 5.6, reports for MORA a 11% increase of utility when providing a service through a set of micro-services [152] running on different con-

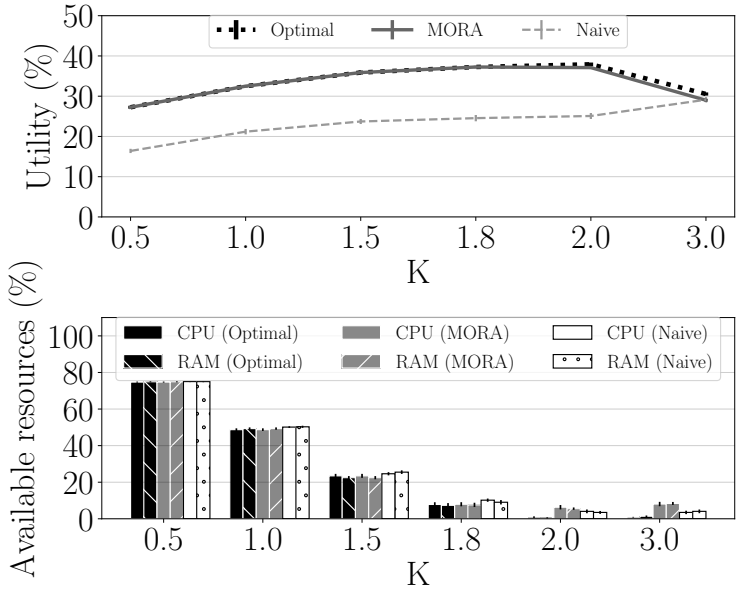

Figure 5.7: Effects of load.

tainers instead of a single one. Indeed, keeping the same overall resource requirements, "smaller" containers are easier to place into Edge nodes.

Effects of load (Fig. 5.7)

Increasing the load K (5.11) means increasing the amount of resource contention among SPs. Recall that $K = 0.5$ denotes requests with overall resource requirements that are half of the available resources. In this case, the highest utility-option of each SP is likely to be satisfied. For $K \leq 2$, the more the K, the more utility, from 28% to 40%. This is expected since (i) increasing the K, we are increasing the resource requirements of each option and (ii) the utility of each option is randomly

generated as an increasing function (5.12) of the resources. However, if the load factor increases more than 2 the utility starts to decrease both for the ILP and MORA and equals the naive policy. Further investigation is required to explain this behavior in our future work. Hypothesis are: (i) there is a concave relation between resources and utilities (see (5.12)), which reflects in the diminishing returns observed when increasing K, and the the resources utilized; (ii) as resources demanded by the containers become larger as K increases, it is more difficult to place them, which is confirmed by the fact that the overall resources used with $K = 2$ and $K = 3$ remain unchanged. From these results, at least in these scenarios, it is observed that after a certain load threshold, the network operator should increase the Edge resources in order to maintain utility levels. As expected, the bottom figure shows that the NO needs to consume more resources to satisfy higher loads.

Distribution of resources and utility among SPs

Also if this work mainly considers that utility benefits NO, in reality it also benefits SPs. Therefore, it is important to check if resource allocation is fair. Fig. 5.8 reports the result from one run with 50 SPs. On the left, the points along each line represent the utility of the different options declared by one SP. The "◆" is the option chosen by MORA. Note that MORA only selects options for 11 SPs, and thus the others do not get any resources since their contribution to the overall utility would not be remarkable. In the right plot, the points are the requirements of the options of the 11 SPs chosen to be deployed in the Edge. From the plot, MORA allocates a similar amount of resources to the selected SPs, which does not necessarily reflect in equality of utilities selected.

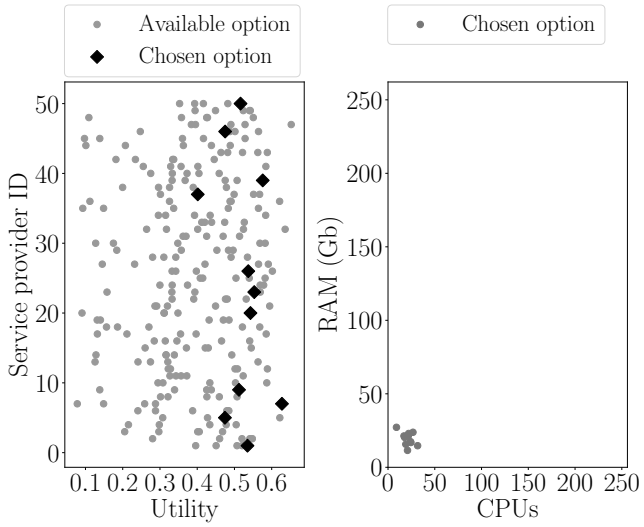

Figure 5.8: Distribution of resources and utility between 50 SPs. The x and y scale of the right plot are the total available RAM and CPUs.

5.4.2 Results on real traces

MORA has been tested against Google [153] and Alibaba [154] cluster traces. In these simulated 8 nodes are available, as in table 5.2.

Google traces includes a list of jobs, each composed by a set of tasks. It is considered the requested RAM and CPU associated to each task when the correspondent at time of job's submission. These values are expressed as a fraction of the available resources. To represent a SP i, the simulator randomly selects J jobs and interprets each as an option i, j. Each task composing that job is mapped to a container z. (5.12) is used to compute the utility of each option, since no network information or other utility parameter is available on these traces.

Alibaba traces include a list of applications, each comprised of several containers sharing the same application index. An Alibaba application,

in the simulation, is mapped to an option. As in the Google case, each SP is randomly selected with a set of random options (Alibaba applications). In Alibaba traces, RAM requirements are expressed as percentage of RAM available in one node while CPU requirements are expressed as percentage of usage of one single core. Each machine in Alibaba cluster has 96 cores. The nodes in the simulations are set to reflect these requirements.

The trace also reports the bandwidth associated to a container. Differently from all other simulations, the bandwidth of each option is calculated as the sum of the bandwidth of the containers of the corresponding application. This represents the value of utility for this scenario. The rationale is to assume that a SP generates a certain traffic toward users. If the algorithm selects a certain option of a SP, the correspondent bandwidth is served locally at the edge and only the remaining part must be served by the Cloud, an assumption common in the literature [71]. Therefore, the value of the utility indicates the bandwidth saving in this case.

Effects of number of SPs

Fig. 5.10 and 5.9 confirm what was observed in the synthetic scenario. Both in the Google and the Alibaba case the utility increase when exploiting service elasticity increasing the number of options per SP. We also vary the number of SPs considered, which result in an increase in the load, which we quantify with $\bar{K}_{\mathrm{CPU}}$ and $\bar{K}_{\mathrm{RAM}}$, reported in the figure and calculated as:

$$\bar{K}_l = \bar{W}_l \cdot N \cdot \bar{Z} / \sum_{m=1}^{M} c_{l,m}$$

where $\bar{W}_l$ is the average requirement of resource l through all the containers and $\bar{Z}$ is the average number of containers per option, l is $\{RAM, CPU\}$. In the Alibaba figure, as expected, the more SPs join the Edge, the more

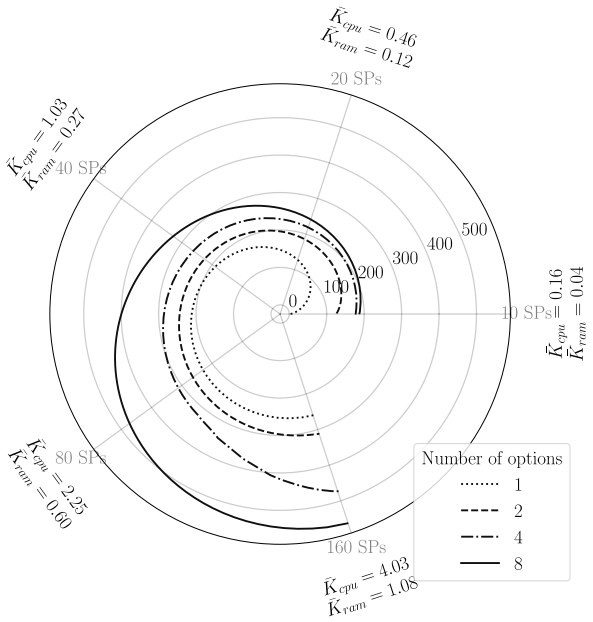

Figure 5.9: Utility while varying the number of SPs and the number of options provided using Alibaba Traces

the utility (bandwidth saving) is achievable . In the Google figure is reported the normalized utility (5.13) (the total utility can be obtained by multiplying it by N). Observe that the values of load K_{CPU} and K_{RAM} are very different. And different are also the Google, Alibaba and synthetic scenarios. However, the benefits of service elasticity consistently show themselves.

MORA as anytime algorithm

By using Alibaba traces and assuming 8 nodes and 20 SPs, it is being shown in Fig. 5.11 the utility of the solution computed at every iteration

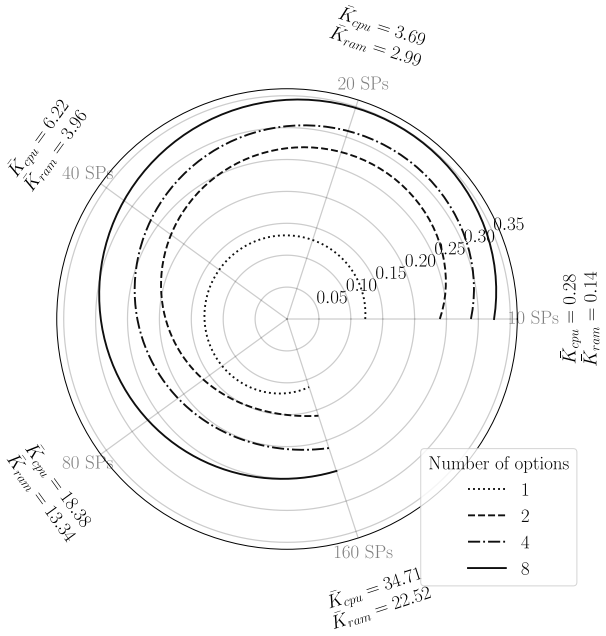

Figure 5.10: Relative utility while varying the number of SPs and the number of options provided using Google Traces

t. By construction, the efficiency E^i of the best jump (bottom plot) is decreasing with t. This ensures that most of the utility is already achieved in the first iterations (top plot) and allows us to prematurely stop the algorithm if the time available to compute the allocation is scarce, still having a good solution at hands. The upper bound to the optimal solution (§5.3.2) reported in the figure also confirms that we are already close to the optimum in the first iterations. By exploiting the fact the monotonicity of E^i we can also easily calculate, at any iteration t, the *Expected utility*, i.e., the utility that can be achieved at most if the algorithm continues until the end instead of interrupting at t.

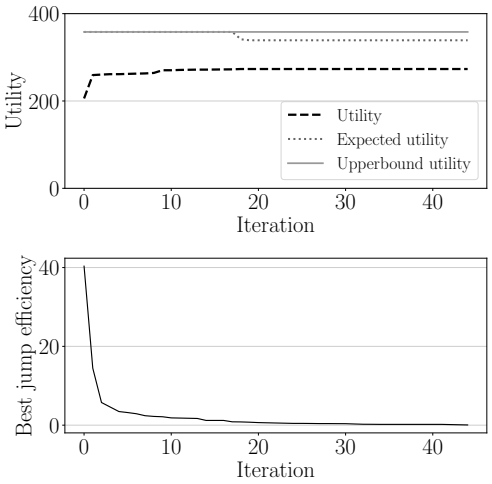

Figure 5.11: Utility and Best jump efficiency by time

5.5 Conclusion

This chapter presented MORA, a strategy for resource allocation for Edge Computing (EC), where tenants are third party Service Providers (SPs). The novelty of this work is that it exploits service elasticity: by allowing SPs to declare the different configurations (aka options) in which they can run, it is shown that the Network Operator (NO) owning EC resources can greatly increase utility. Relying on service elasticity is crucial in resource-constrained environments as EC.

A future work will be devoted to scenarios where jobs arrive in different times, exploiting a time-batched and online implementation of MORA.

Finally, as anticipated in the introduction of this part, these two works can be considered a way to open a window on the importance of

communication optimization for networks of containers, both in cloud, multi-cloud and edge/cloud scenarios. Future work will be devoted to the evaluation of the strategy in chapter 4 as the internal, network-aware, scheduler for a set of clusters, with a focus on the inter-cluster communication performance by exploiting characteristics of MORA. Last but not least, future work will explore the possibility to enable edge roaming capabilities to optimise how requests, e.g., in a locality, are routed through multiple dislocated set of nodes at the edge up to the local nodes exposing elastic services, as in this chapter, and the cloud.

Part III

Towards AIOps: Ananke as an orchestration framework and decision support system

Chapter 6

Introduction

> *« So my aim here is not to teach the method that everyone must follow for the right conduct of his reason, but only to show in what way I have tried to conduct mine. »*
>
> *René Descartes*

DevOps philosophy aims to reduce the gap between Developers and Operations teams, enabling automation, integration, monitoring, and collaboration by exploiting continuous integration and deployment [27, 29, 155, 156]. DevOps is essential to guarantee release consistency and time-to-market deadlines. Netflix declared [157] to perform around 90 deployments per day, which go into production in a time window of 16 minutes after a commit in their version control system.

In [158], the authors report that companies, adopting DevOps, perform 46 times more frequent code deployment, 440 times faster Change Lead Time and 170 times faster Mean Time to Recovery.

Currently, different companies and researchers adopt DevOps on different scenarios like Internet of Things [159], Cloud-applications [160], Streaming services [161], leveraging different architectures [159, 162], tools [118, 163–169] and/or methodologies [156, 162, 170].

There is a need to improve collaboration between resource providers and Application providers [1, 5].

In 2019 [30], Gartner came out with the "AIOps" keyword, inviting companies and researchers to combine DevOps methodologies and requirements with BigData, Artificial Intelligence (AI), Machine Learning (ML), and Deep Learning.

There is no widely accepted definition for AIOps yet. In [31], the authors define AIOps as techniques to empower software and service engineers to efficiently build, deploy and maintain services using artificial intelligence techniques. Challenges for AIOps research goes towards reducing operational cost [82], improving productivity, customer satisfaction [3, 4].

However, as stated in [31], AIOps solutions in real-world scenarios are still challenging: there is a need for covering the gap in innovation methodologies leveraging highly-skilled engineers and researchers from different fields.

Traditional monitoring strategies and their produced data are not usually compatible with deep learning techniques (e.g., supervised learning based on labeled data is not straightforward with available data from current standard platforms).

Monitoring and modeling Cloud-Native Applications (CNAs), getting high-quality data become a key to sort together real-world applications performance metrics and optimization strategies involving Big Data and AI/ML.

In analogy with control theory, monitoring capability of a CNA can be considered as the complex systems "observability" property while optimizing and supporting operations lay in the field of "controllability". We need to evaluate errors and accuracy through a monitoring system (the

observer) to guarantee controllability. Typically, however, this modifies the system itself, and we need to minimize system-observer interference.

Analogously, distributed systems require (i) a monitor which minimally interferes within the observed CNAs, and (ii) controllers to minimize errors and drive orchestrators through decision support, anomaly detection, etc.

The behavior of these distributed applications are strictly related (i) to the Quality of Service (QoS) (mainly the performance, availability, and robustness) of their components, which are often microservices, (ii) to the way they are deployed, (iii) the hardware resource needed to run them adequately, and (iv) to the network connecting them.

Each SLO of an SLA may involve different SLIs which specify different units and scales depending on the application scenario.

The SLAs define the performance requirements that the applications have to satisfy while running in terms of delivery times, throughput, fault tolerance, and high-availability. Beyond the impact on performance of the applications, violating SLAs will cause a business and economical dispute between a resource provider and a customer. During the application runtime, fluctuations in its behavior may arise from an unexpected or excessive workload, hardware failures, or networking issues, affecting - sometimes for a prolonged time - its ability to perform the application tasks with the appropriate level of quality and thus compromising the SLA requirements.

When this happens, a set of corrective operations must be carried out to bring the application back to the desired state: who decides which operations must be selected, how and when to execute them, characterize the type of control system of the application. Today's research focuses on supporting human operator decisions through AI-driven, au-

tonomous, control systems, directly integrated with the applications or their execution environment.

In this part, *Ananke* is presented as a model and architecture to both monitor CNAs and analyze those time-series data, modeled as a time-varying multi-layer network [171, 172].

Chapter 7 will focus on the tracing model and the Ananke architecture that will be used from now on to present the strategies for assessing AIOps strategies that aim to optimize the orchestration of cloud-native applications.

By means of providing both white-box (fine-grained) and black-box monitoring, the architecture of *Ananke* exploits Prometheus instances for time-based metrics and Open Tracing based SPOUTs on pulling specific metrics as provided by the application providers and as an input for the Raphtory graph processing system [2, 173].

Chapter 8 deep dive into the black-box monitoring capabilities of *Ananke*, providing design and implementation of the tools that enable interaction between data collected from the cluster and applications through the *Ananke* Prometheus instances, and the AIOps analysers that will perform decision support and control of the cluster. In particular, it reports the design and implementation of (i) a Prometheus Backfilling library to bulk import metrics from legacy monitoring systems, and (ii) a Python library that interact with Prometheus using data structures and third-party tools typical of ML/DL pipelines (i.e., Pandas DataFrames, Tensorflow, ...), and publishes results of the analysis over generic network channels where actuators of the system can subscribe in order to smart orchestrate the system.

Finally, chapter 9 presents NAPA, a strategy that applies a reinforcement learning (RL) approach taking into account the trends of the hardware resources utilization, and the risk of not fulfilling the SLAs.

In particular, an RL agent is added to the pipeline given by the FBProphet-based forecasting in chapter 8 and, thus, to choose an action that minimise the risk of violating the SLA.

When the risk to violate SLAs becomes high for a given application, the proposed approach can efficiently decide whether to scale-out the micro-services deploying new replicas or to execute an adaptive algorithm that supplies an autonomic dynamic routing leading to better exploitation of the Flannel-based software-defined network consisting of both the micro-services, and the inter-workers network.

Chapter 7

Towards AIOps: Ananke

«The best and safest method of philosophizing seems to be, first to enquire diligently into the properties of things, and to establish these properties by experiment, and then to proceed more slowly to hypothesis for the explanation of them.»

Isaac Newton

As already discussed in previous chapters, micro-service architecture enables smarter management of applications life-cycle. However, the increasing of the number of components also increases complexity, especially on operations like migration and horizontal scaling. Operations in micro-services based applications can get complex and should involve parameters and properties like connection throughput, resources usage, robustness or consistency and reliability. To perform this kind of operations and optimization strategies in micro-services applications, this chapter exposes the work done on defining Ananke, a framework consisting of a time-varying multi-layer graph-based model and architecture to profile micro-services and their interactions in a platform-as-a-service environment. The aim is to provide support and facilities for optimization strategies that a Cloud Provider can exploit to guarantee quality

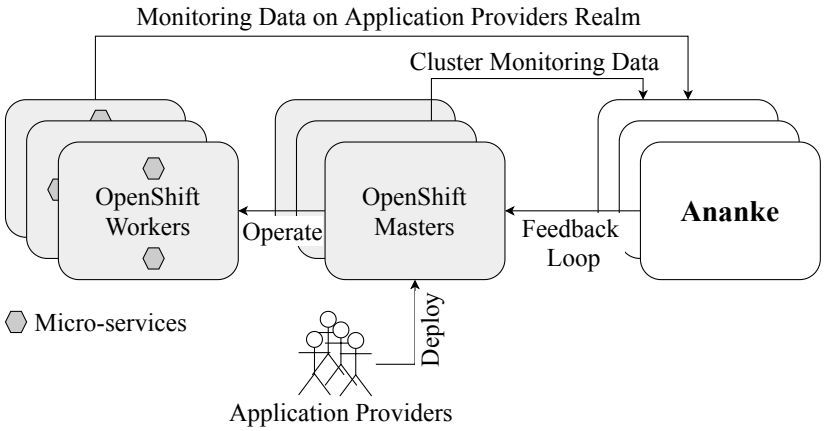

Figure 7.1: Reference scenario

of service and service-level agreements. Next chapters will make use of this model to present strategies for the auto-scaling of micro-services and the reconfiguration of underlying network in an adaptive algorithm exploiting reinforcement learning.

Let's consider a Resource Provider (RP) owning a cluster of heterogeneous virtual machines (VMs) logically partitioned into M_w workers and a different set of masters, responsible for the orchestration. Finally, a set of VMs is responsible for monitoring. This cluster, depicted in Fig. 7.1, is located within a single Data Center and is based on Red Hat OpenShift [111]. The RP's resources are shared among its customers, the application providers (AP), which ask to deploy their workloads as usual through the RP tools.

Despite the RP's ability to configure its facilities and monitoring cluster metrics considering APs' workloads as black-boxes, it is challenging to deep dive into those workloads and monitor service-level indicators in order to guarantee SLAs.

Ananke is being proposed as a collector of both Cluster metrics and internal monitoring of single applications and micro-services.

Like similar systems, e.g., Amazon CloudWatch [174], *Ananke* requires that the APs configure their code to communicate with the monitor; however, in the case of *Ananke*, the collected metrics are for the use of the RP itself, and data model has to be consistent across applications.

Ananke also has to consider the security issues derived from knowledge sharing between the RP and the APs, and it has to avoid any possible leak of information between different administrative domains.

In the following, objects abstractions typical of Kubernetes (K8S) and OpenShift orchestration dialect are being used to be synthetic (i.e., Pods, Services, ...). Although OpenShift is the orchestrator to which Ananke refers, this choice does not represent a limit in the generality of the proposed approach.

7.1 Data Model

7.1.1 Cluster Model

We start by defining the cluster in which the applications run. In particular, worker nodes can be considered a fully connected graph. In the following, $\mathbb{U} = \mathbb{R} \cup \mathbb{A}^*$ represents the universal set of both real numbers and strings of alphabet $\mathbb{A}$ (e.g., ASCII). Moreover, cartesian products have to be considered flattened (page 80 of [175]).

Definition 8. *Given an RP Cluster C with M_w workers, we define the multi-label fully-connected network associated with C, at a specific time $\tau \in \mathbb{N}$, as:*

$$\mathcal{G}_{C,\tau}(\mathcal{M}_w, \mathcal{M}_w^2, P_\tau^w(w, k_w), P_\tau^l(v, w, k_l)) \tag{7.1}$$

where:

- $\mathcal{M}_w$ *is the set of vertices;*

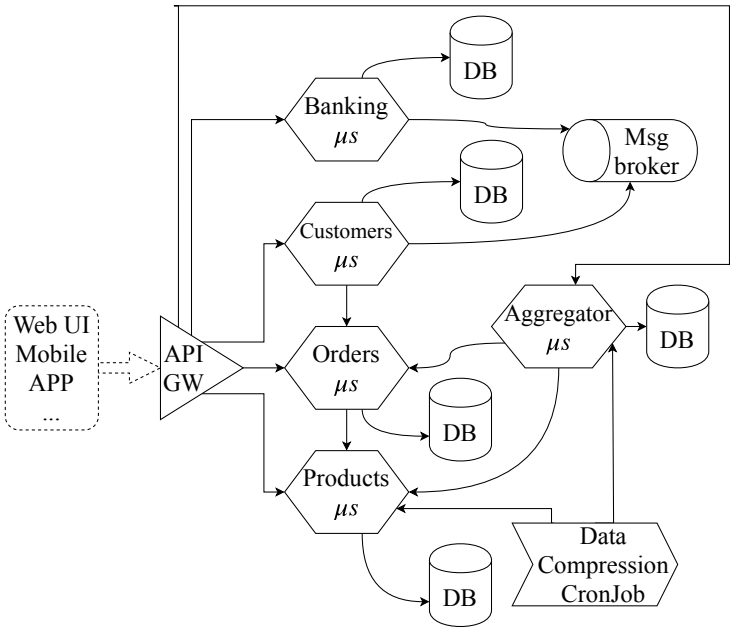

Figure 7.2: Example of CNA

- $P_{\tau}^{w}: \mathcal{M}_w \times \mathcal{K}_w \rightarrow \mathbb{U}$ *is a function that maps a key in the set* $\mathcal{K}_w$*, for a given vertex* $w \in \mathcal{M}_w$*, to the value of the property (e.g., CPU/Mem usage);*
- $P_{\tau}^{l}: \mathcal{M}_w^2 \times \mathcal{K}_l \rightarrow \mathbb{U}$ *is a function that maps a key from the set* $\mathcal{K}_l$*, for the given edge between vertices* $v \neq w \in \mathcal{M}_w$*, to the value of the property (e.g., network bandwidth).*

In table 7.1, examples of keys for the sets $\mathcal{K}_w$ and $\mathcal{K}_l$.

7.1.2 Deep dive into micro-services monitoring

As shown in fig. 7.2, a micro-services-based CNA can be considered as a set of de-coupled units that interact with each other through either synchronous protocols such as *HTTP/REST* and *RPC* or asynchronous

Table 7.1: Reconstruction keys and example of Vertex/Edge property keys

Graph reconstruction keys	**Description**
X-Request-ID $\rightarrow t$	Unique ID for a given request type instance
X-Root-ID $\rightarrow R \in \mathcal{V}_{A,r}$	Root of a path
X-Caller $\rightarrow (\boldsymbol{v}, w) \in \mathcal{E}_{A,r}$	It is the micro-services that sent the request to the current service
X-Service-Name $\rightarrow (v, \boldsymbol{w}) \in \mathcal{E}_{A,r}$	The name of the micro-services in which the request arrives
X-Req-Type $\rightarrow r \in \mathcal{R}_A$	A unique identificator of a *request type* (e.g. method + endpoint on a HTTP micro-services.)
micro-services vertex properties ($\mathcal{K}_v$)	**micro-services edge properties ($\mathcal{K}_e$)**
Start/End time	Bytes tx
Response status (e.g. 200 OK, Query Error)	Bytes rx
Type of action (e.g. HTTP GET, SQL SELECT...)	Bandwidth tx
Executor (e.g. IP/MAC)	Bandwidth rx
Worker vertex properties ($\mathcal{K}_w$)	**Worker edge properties ($\mathcal{K}_l$)**
CPU Usage	Bytes tx
RAM Usage	Bytes rx
Load factor	Bandwidth tx
Hostname	Bandwidth rx

protocols via message brokers. Typically, each service is associated with its database. In a CNA, the API Gateway is the actual entry point for

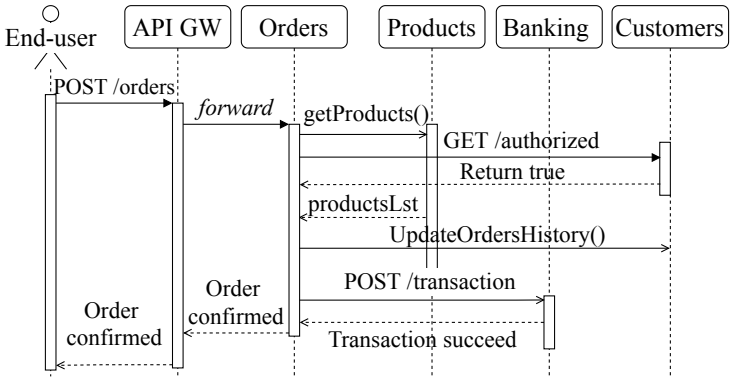

Figure 7.3: Example sequence diagram of an action performed in a CNA

external clients; it is usually the component that begins the chain of calls to be performed due to a client request. We call this chain a *path*.

However, other specific micro-services of a CNA deployment can trigger actions and initiate a *path* within the application architecture.

In the example CNA in Fig. 7.2, a micro-services, *"CronJob for DataCompression"*, can initiate a path involving different interactions with other micro-services. Another micro-services in the example is the *"Aggregator micro-services"*, which points out how interactions within a CNA can be arbitrarily complex and give deeper chains. Fig. 7.2 gives a deployment view; it is static in the sense that no quantitative information can precisely be provided for any kind of interaction between micro-services. For example, a POST request to an endpoint in a REST API usually involves one or more *updates* into databases, while a GET can involve one or more *reads*; these two *request types* can be performed by the same micro-services, but they may have different performances (e.g., response/processing times) and different occurrence during the application life-cycle. An example of an interaction between micro-services is reported in the sequence diagram of Fig. 7.3. Here, the kind of infor-

mation is deeply related to the single request type (i.e., HTTP POST */orders* to the API Gateway).

While Fig. 7.2 reports the deployment view, consisting of all the micro-services that interact within the CNA during the whole life-cycle of the application, the sequence diagram in Fig. 7.3 exposes just a subset of the micro-services, the ones which interact during a given activity. However, it also reports the evolution in time of the performed action, the *chain*, but it is still static: no run-time information is provided.

Ananke aims to join information from both those two views in a dynamic one, which involves information by the run-time environment.

Moreover, replicas have to be considered: considering micro-services like K8S *services* is useful to redact them from the graph model; each micro-services of the same service (a replica Pod) is an executor uniquely identified in the virtual network, but the most important information can be aggregated into a unique *service* abstraction. Wherever needed, we can still retrieve replica-specific information through properties like "executor IP/MAC".

To proceed, $t \in \mathbb{N}$ is the unique identifier of a request instance (e.g., a specific call to the request type "POST /orders" to the API Gateway). In particular, t is paired to the field "X-Request-Id" of the data retrieved by the applications' monitors, as reported in table 7.1. t is also considered unique among all the applications and all their request types.

In the upper part of Fig. 7.4, an example of the graph that defines paths. Looking at the left path (example of a path for application 1), vertices v_* represent the micro-services involved in the specific action. In this case, v_6 is annotated as the root vertex, the one who receives the trigger to initiate the action. Then, the edges represent the interactions between micro-services, and the "documents" P^* are the lists of properties related to the edge or to the vertex (we report just two of

these lists, for clearness). Note that the graphical representation of path reports just a "snapshot" of the specific instance (t_1 for application 1, t_2 for application 2).

Definition 9. *Let us consider an application A consisting of a set of micro-services $\mathcal{M}_A$. Given a request type r and one of its instances, identified by t, we define the* path $p_{A,r,t}$ *as the multi-label directed rooted graph*

$$p_{A,r,t}(\mathcal{V}_{A,r}, \mathcal{E}_{A,r}, R_{A,r}, P^v_{A,r,t}(v, k_v), P^e_{A,r,t}(v, w, k_e)) \tag{7.2}$$

where:

- $\mathcal{V}_{A,r} \subseteq \mathcal{M}_A$ *is the set of vertices representing the micro-services involved in r;*
- $\mathcal{E}_{A,r}$ *is the set of edges (tuples in $\mathcal{V}_{A,r}$) which represent the interaction between micro-services for r; they are directed from the caller micro-services to the receiver one;*
- $R_{A,r} \in \mathcal{V}_{A,r}$ *is the root of the graph: the micro-services which triggers interactions;*
- $P^v_{A,r,t} \colon \mathcal{V}_{A,r} \times \mathcal{K}_v \to \mathbb{U}$ *is a function that maps a key in the set $\mathcal{K}_v$, for a given vertex $v \in \mathcal{V}_{A,r}$, to the value of the property (e.g. response time);*
- $P^e_{A,r,t} \colon \mathcal{E}_{A,r} \times \mathcal{K}_e \to \mathbb{U}$ *is a function that maps a key in the set $\mathcal{K}_e$, for a given edge $(v, w) \in \mathcal{E}_{A,r}$, $v \neq w$, the value of the property (e.g. rx/tx bytes);*

The first section of Table 7.1 reports the mapping between data retrieved in the monitor time-series and the entities of the above defined graph, while the second part reports examples of the sets K_v and K_e.

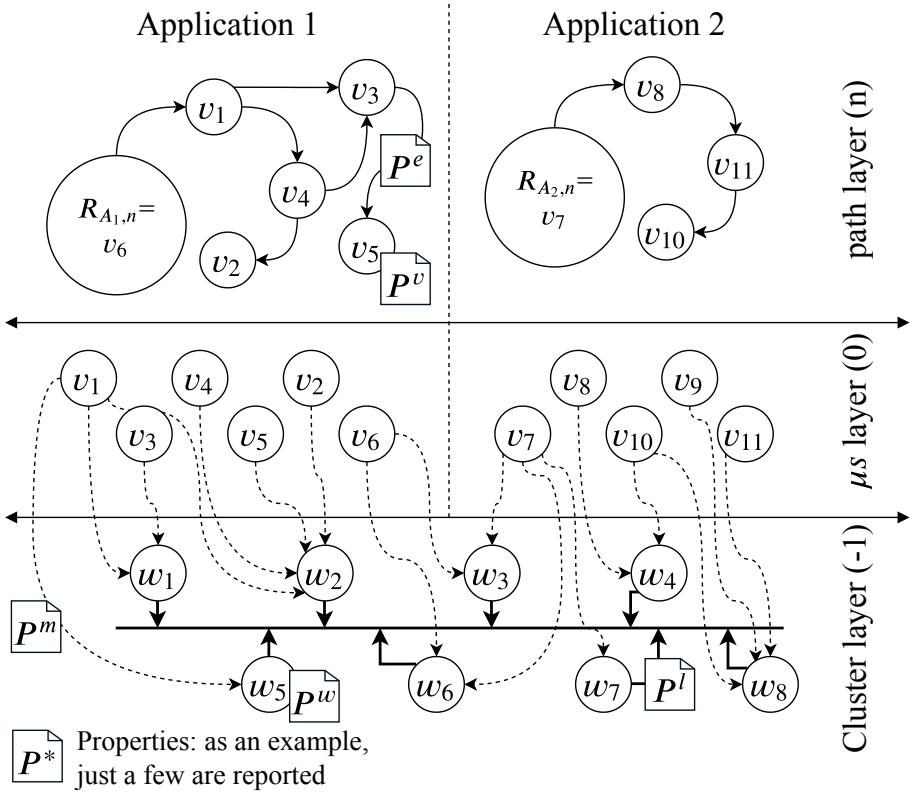

Figure 7.4: Example of graph $\mathcal{G}_\tau$ in eq. (7.3) for a cluster of 8 workers and 2 applications. Only 1 instance of 1 path per application is reported: path n of A_1 at instance $t_1 \in \mathcal{T}_\tau$ and path n of A_2 at instance $t_2 \in \mathcal{T}_\tau$.

7.1.3 Application & Performance Model

As defined in section 7.1.2, each action that can be performed either from an external source (e.g., end-user) or from an internal trigger (e.g., CronJob) creates a path that can be formally described as the graph in def. 9.

Therefore, we can consider an application as a "set" of paths, a multiplex network [172].

In particular, let's call:

- $\mathcal{R}_A$, the set of request types for application A;
- $\mathcal{V}_A = \bigcup_{r \in \mathcal{R}_A \cup \{0\}} \mathcal{V}_{A,r} \times \{r\}$, the set of vertices associated with their request type: a tuple (v, r), where v is a micro-services, r is a request type that identifies a path; the $(v, 0)$ tuple is used to represent an auxiliary layer, an edge-less graph with all the micro-services.
- $P^v_{A,t} \colon \mathcal{R}_A \times \mathcal{V}_A \times \mathcal{K}_v \to \mathbb{U}$ is a function that returns the value of property $k_v \in \mathcal{K}_v$ associated with request type $r \in \mathcal{R}_A$ for vertex $v \in V_{A,r}$ of path $p_{A,r,t}$. Which is $P^v_{A,t}(r, v, k_v) = P^v_{A,r,t}(v, k_v)$;
- $P^e_{A,t} \colon \mathcal{R}_A \times \mathcal{E}_A \times \mathcal{K}_e \to \mathbb{U}$ is a function that returns the value of property $k_e \in \mathcal{K}_e$ associated with the request type $r \in \mathcal{R}_A$ for edge $(v, w) \in \mathcal{E}_{A,r}$ of path $p_{A,r,t}$. Which is $P^e_{A,t}(r, v, w, k_e) = P^e_{A,r,t}(v, w, k_e)$.

By exploiting the defined graphs and entities, information on a single path $p_{A,r,t}$ can be obtained. In particular, a *super-path* is referred as all the instances of a request type, i.e., $p_{A,r,*}$; considering a super-path is useful to get information about the performance of a given request type. Therefore, to get information about the whole application, it is needed to aggregate information from all the super-paths of the application, i.e., $p_{A,*,*}$. Finally, information from the application, i.e., $p_{A,*,*}$, have to be paired with information from the cluster $G_{C,\tau}$, by mapping micro-services graph to the cluster graph, to know in which worker a micro-

services is deployed, and each instance t of a path to the actual time τ. Formalization of this relationship is given in the following.

7.1.4 Putting it all together

Let's define $\mathcal{A}$ the set of applications hosted by an RP. By using def. 8 and 9, a multi-layer network can be built. Different graphs, as already shown, are involved: the application graphs ($p_{A,*,*} \forall A$), the cluster graph ($\mathcal{G}_{C,\tau} \forall \tau$), and their associated properties. In the multi-layer network, one layer (identified with -1 in the next) is associated with the cluster and an auxiliary layer (identified by 0) is used to represent the the edge-less graph of all the vertices, i.e., micro-services, of any application $A \in \mathcal{A}$. The abstraction provided by layer 0 is useful considering inter-layer edges between a micro-services v and all the workers w in which orchestrator deploys replicas of v.

Let us assume that all the micro-services are neither scaled nor migrated in a ϵ-neighborhood of τ.

We can define those inter-layer edges at time τ as a set $\mathcal{E}_{m,\tau}$ of 4-tuples. In particular, given a replica of a micro-services v hosted on worker w, the inter-layer edge is represented as a 4-tuple $(v, w, 0, -1)$.

As an example, look again at fig. 7.4: the whole figure represents the multi-layer network at time τ. The bottom layer (-1) is the layer of fully-connected cluster network from def. 8; the dashed edges from w_* vertices to the v_* vertices in the *micro-services layer* are the inter-layer edges just defined above. Finally, the upper layer is just one of the paths layers. That figure still represents the snapshot of the time-varying multi-layer network at time τ but should include all the paths that "lived" (executed) in the ϵ-neighborhood of τ.

In fact, t is a unique identifier to differentiate single instances of paths in Def. 9, while in Def. 8, τ is the actual time to get the time-series of cluster performances.

Now, as already introduced, an instance t of a request type r, the path $p_{A,r,t}$, executes at a specific time, and t must be paired to the actual time τ to get a complete view of applications' performances.

To associate t and τ, the properties StartTime and EndTime of paths can be exploited, deriving the set of paths that executed in the given ϵ-neighborhood of τ. The following definition resumes all the considerations of this section:

Definition 10. *Let us consider a cluster C and a set of applications $\mathcal{A}$. Let $\tau \in \mathbb{N}$ be a specific time. Let T^* be the set of all t. Let $\mathcal{T}_\tau$ be:*

$$\mathcal{T}_\tau = \{t \in T^* : |P^v_{A,t}(r, v, startTime) - \tau| < \epsilon \vee$$

$$|P^v_{A,t}(r, v, endTime) - \tau| < \epsilon, v = R_{A,r} \ \forall A \in \mathcal{A}, \forall r \in \mathcal{R}_A\}$$

$\mathcal{T}_\tau$ is the set of all requests instances that were running in the ϵ-neighborhood of τ. The graph $\mathcal{G}_{C,\tau}$ in (7.1) represents the cluster. Each application $A \in \mathcal{A}$ is represented by all the paths associated with its request types $\forall t \in \mathcal{T}$.

We define the multi-label rooted multi-layer network associated with the whole system $(C, \mathcal{A})$ at time $\tau \in \mathbb{N}$ as:

$$\mathcal{G}_\tau(\mathcal{L}, \mathcal{V}, \mathcal{E}_\tau, P_\tau(v, w, l, m, k, t), R(A, r)) \tag{7.3}$$

where:

- $\mathcal{L} = \{-1, 0\} \cup \bigcup_{A \in \mathcal{A}} \mathcal{R}_A$ *is the set of layers, -1 is the cluster layer, 0 is the layer of the applications micro-services;*
- $\mathcal{V}$ *is the set of vertices of both cluster and applications:*

$$\mathcal{V} = \mathcal{M}_w \times \{-1\} \cup \bigcup_{A \in \mathcal{A}} \mathcal{V}_A \tag{7.4}$$

- $\mathcal{E}_\tau$ *is the set of edges:*

$$\mathcal{M}_w^2 \times \{-1\}^2 \ \cup \mathcal{E}_{m,\tau} \ \cup \bigcup_{\substack{A \in \mathcal{A} \\ r \in \mathcal{R}_A}} \mathcal{E}_{A,r} \times \{r\}^2 \tag{7.5}$$

- $P_\tau \colon \mathcal{E}_\tau \times (\bigcup_{i \in \{v,e,w,l\}} \mathcal{K}_i) \times (\mathcal{T} \cup \{0\}) \to \mathbb{U}$ *is a function which maps the value of property key* k *for edge* $(v, w, l, m) \in \mathcal{E}_\tau$ *between vertices* $v \neq w \in \mathcal{V}$ *with* v *in layer* l *and* w *in layer* m*. If* $v = w$ *and* $l = m$*,* $P_\tau(\cdot)$ *returns the value of property* k *for vertex* v*. If* $t = 0$*, it returns a property of the* -1*-th layer (the worker data at actual time* τ*).*

$$P_\tau = \begin{cases} P_{A,t}^v(r, v, k) & k \in \mathcal{K}_v, \\ & v = w \in \mathcal{V} \setminus \mathcal{M}_w, \\ & r := l = m \in \mathcal{L} \setminus \{-1, 0\} \\ & A := a \in \mathcal{A} | r \in \mathcal{R}_A \\ P_{A,t}^e(r, v, w, k) & k \in \mathcal{K}_e, v \neq w, \\ & r := l = m \in \mathcal{L} \setminus \{-1, 0\} \\ & A := a \in \mathcal{A} | r \in \mathcal{R}_A \\ P_\tau^w(w, k) & k \in \mathcal{K}_w \\ & l = m = -1, t = 0 \\ & v = w \in \mathcal{M}_w \\ P_\tau^l(v, w, k) & k \in \mathcal{K}_l \\ & l = m = -1, t = 0 \\ & v \neq w \in \mathcal{M}_w \\ P_\tau^m(v, w, l, m, k) & l = 0, m = -1, k \in \mathcal{K}_w \end{cases} \tag{7.6}$$

with P_τ^m *representing the value of a property key in* $\mathcal{K}_w \cup \mathcal{K}_l$*, for the given micro-services replica* v *hosted on worker* w*;*

- $R \colon \mathcal{A} \times (\bigcup_{A \in \mathcal{A}} \mathcal{R}_A) \to \mathcal{V}_\tau$ *returns the root vertex* $R_{A,r}$ *of path* $p_{A,r,t}$*.*

With no lack of generality, applications considered till now do not interact with external services.

7.2 Architecture and workflow

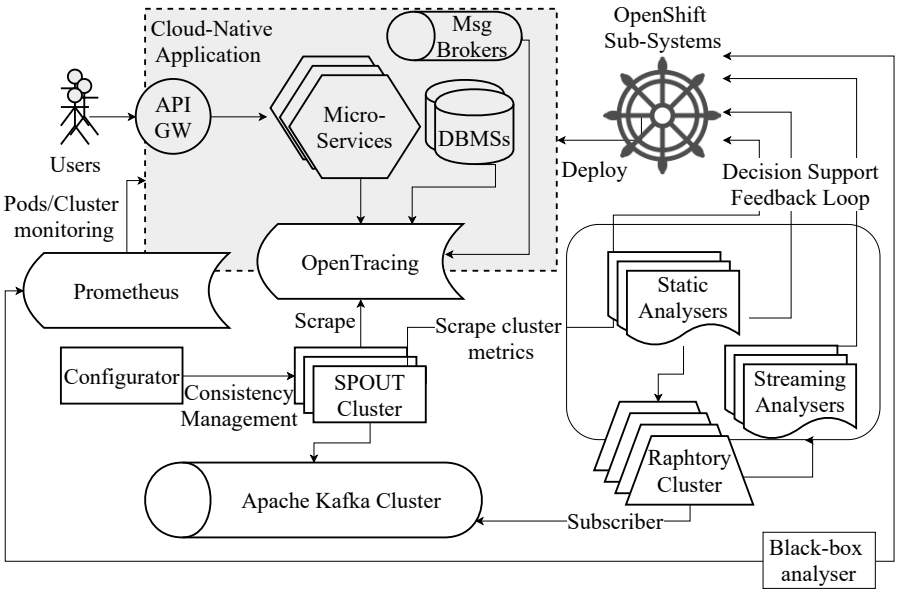

Figure 7.5: *Ananke* architecture with focus on white-box monitoring

The data model defined in the previous section allow to build a fine-grained view of applications' performance, their internal interactions, and the relationship with the workers on which orchestrator deploys applications' micro-services. By monitoring, collecting and processing these data in a time-varying multi-layer network, *Ananke* can identify anomalies, automatically fix run-time issues, or optimize resource allocation, QoS. In Fig. 7.5 the software architecture to exploit this data model and consequent analyses and actuators. *Ananke* components are

also based in the micro-services pattern, which can be described in three main classes: (i) monitor & collect, (ii) storage & processing middleware, (iii) analysers and actuators. As in Fig. 7.5, the components associated with monitoring and collection of time-series data lay in the border between running applications (with their associated namespaces/projects in K8S/OpenShift abstractions) of APs and the RP monitoring layer.

Ananke relies on OpenTracing APIs [176] and a graph processor for the white-boxing monitoring, while Prometheus is responsible of the time-based black-box monitoring of the cluster and the hosted applications.

A library is able to provide facilities to configure code quickly through decorators.

The kind of metrics exchanged by the micro-services is the ones provided, for example, in Table 7.1.

Those metrics are scraped by a cluster of SPOUTs. The SPOUT is a Python micro-services: it is a stateless component that relies on the "Configurator" for consistency, partitioning, and reliability configuration.

Any SPOUT is in charge of a sub-set of the system $(C, \mathcal{A})$. SPOUTs, iteratively, (i) get last data from the monitors, (ii) adapt them to the entities defined in the previous section, and (iii) publish these entities through Kafka to the streaming graph processing system.

Raphtory [2, 173] is a distributed temporal graph management system. It is able to maintain the full graph history in memory, providing: eventual consistency, snapshots-based persistence and fault-tolerance. It is configured as a subscriber of the Kafka Queue, and enables analysers to perform both analyses on the most up-to-date version of the graph, and temporal analysis throughout its full history. Analysers, finally, implement the Orchestrator APIs: their analyses, therefore, close the feedback

loop: they enable controlling the orchestration. *Ananke* is currently a work-in-progress publicly available as a Free Software on GitHub [13].

7.3 Conclusion

This chapter presented *Ananke*, a framework for monitoring and operating on micro-services-based applications. The proposed approach is based on a time-varying multi-layer modeling of the micro-services, applications, and resource capabilities. Even if *Ananke* focuses on a scenario based on cloud-native and micro-services-based applications, this framework can also be adapted to a vast set of different environments, as for instance, function-as-a-service and metal-as-a-service.

The next chapters will make use of the data model here proposed to build case studies that explore the field of smart orchestration for cloud-native applications by the use of AI/ML tools and strategies in compliance with the paradigm defined by Gartner under the term AIOps.

Chapter 8

Predicting peek events through Facebook Prophet and Scale-out CNAs

«It is not enough to possess a good mind; the most important thing is to apply it correctly.»

René Descartes

In previous chapter, a white-box monitoring approach is mainly considered, in which the APs have to configure their code leveraging a library that is able to communicate with the *Ananke*'s monitoring tools.

This chapter extends the *Ananke* framework, providing a black-box monitoring and decision supporting framework based on Prometheus and, as a case study an auto-scaling strategy based on the prediction of traffic peeks for a web application exploiting the Facebook Prophet model.

Based on the Ananke monitoring model, it investigates design issues and strategies required to enable integration between clusters and appli-

cations managed by Ananke and their metrics stored in the Prometheus Monitoring system. Algorithms for anomaly detection and forecasting of time series have been introduced in the proposed AIOps Prometheus Framework for the system analysis and orchestration of applications.

In particular, as in 7.1, for a given RP Cluster C with M_w workers, and an application A, the multi-label fully-connected network associated with C and the multi-label fully connected network associated with A, at a specific time $\tau \in \mathbb{N}$, are:

$$\mathcal{G}_{C,\tau}(\mathcal{M}_w, \mathcal{M}_w^2, P_\tau^w(w, k_w), P_\tau^l(v, w, k_l)) \tag{8.1}$$

$$\mathcal{G}_{A,\tau}(\mathcal{V}_A, \mathcal{V}_A^2, P_\tau^A(w, k_w), P_\tau^l(v, w, k_l)) \tag{8.2}$$

where:

- $\mathcal{M}_w$ is the set of workers VMs;
- $\mathcal{V}_A$ is the set of pods;
- $P_\tau^w \colon \mathcal{M}_w \times \mathcal{K}_w \to \mathbb{U}$ is a function that maps a key in the set $\mathcal{K}_w$, for a given vertex $w \in \mathcal{M}_w$, to the value of the property (e.g., CPU/Mem usage);
- $P_\tau^l \colon \mathcal{M}_w^2 \times \mathcal{K}_l \to \mathbb{U}$ is a function that maps a key from the set $\mathcal{K}_l$, for the given edge between vertices $v \neq w \in \mathcal{M}_w$, to the value of the property (e.g., network bandwidth).
- $P_\tau^A \colon \mathcal{V}_A \times \mathcal{K}_A \to \mathbb{U}$ is a function that maps a key in the set $\mathcal{K}_A$, for a given vertex $V \in \mathcal{V}_A$, to the value of the property (e.g., CPU/Mem usage for the specific pod);
- $P_\tau^l \colon \mathcal{V}_A^2 \times \mathcal{K}_l \to \mathbb{U}$ is a function that maps a key from the set $\mathcal{K}_l$, for the given edge between vertices $v \neq w \in \mathcal{V}_A$, to the value of the property (e.g., network usage of a pod).

Different useful black-box monitoring information can be obtained either exploiting features of the underlying layers of the OS stack or by facilities on the main components of the applications (e.g., api gateways logs).

As in fig. 7.5, the components associated with monitoring and collection of time-series data lay in the edge between running applications (and their associated namespaces/projects in K8S/OpenShift abstractions) of APs and the RP monitoring layer.

This chapter will focus on the approach for data collection and analysis of the black-box time-series stored in the Prometheus instances of *Ananke*.

Finally, it will provide a case study that aims to predict peek events in a cluster to decide, before the actual peeks occur, either to scale out the related pods or not.

8.1 Data Back-filling on Prometheus

When migrating an application to a cluster managed by *Ananke*, the application providers and the resource provider may need to ingest historical data previously collected by the previous monitoring system in order to allow, as an example, faster training of the analysers. Till now, Prometheus does not allow an efficient way to back-fill metrics from other systems: users' and developers' issues asking for this feature are open on the public Prometheus repository since 2015 [177]. Only recently, the Prometheus team developers are beginning to implement ways to bulk import historical data: they provide a command line tool to import historical data that are already available in OpenMetrics format [178]. Historical data, however, may come from a different monitoring system, and converting them into OpenMetrics is not so trivial.

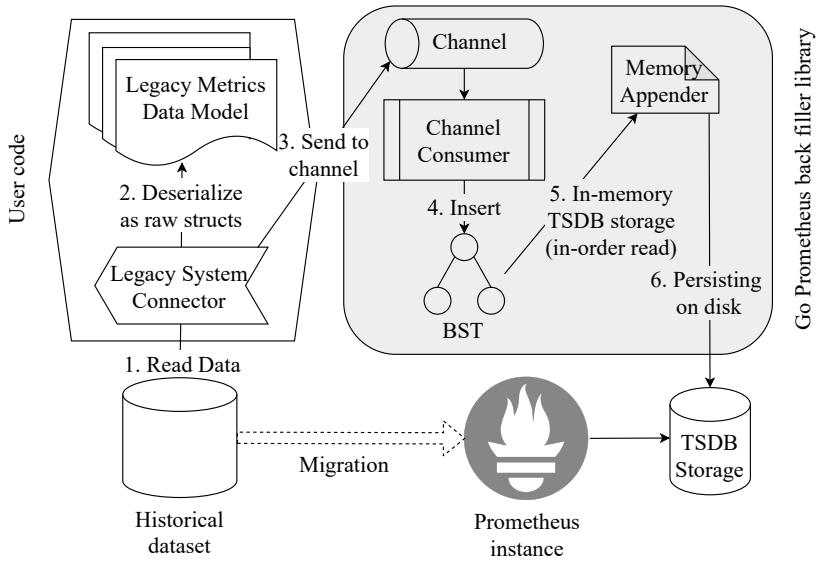

Figure 8.1: Prometheus Back-filler

The solution proposed is a Go library to adapt data from another system and directly store them in TSDB. It is available on GitHub as a Free Software [11]. It allows to define the suitable connectors to a legacy monitoring system (e.g., based on MySql, Parquet/Csv files, HTTP APIs...) and the models representing the metrics to convert into TSDB format. Then, by using data structures of the Prometheus SDK and Go reflection, it will adapt the metrics from the old system and save them into a folder as TSDB blocks that Prometheus can easily read.

The Go Prometheus Backfiller library is based on a queue-based producer-consumer algorithm, with the consumer that (i) receives chunked data from the old system adapter streamed in a Go channel and, (ii) inserts them in a Binary Search Tree (BST); To not slow down insertion in the sorted tree (Avg: $O[\log(n)]$, Worst case: $O(n)$), after reaching a threshold of maximum elements per tree, a separate thread,

traverses the BST in-order and store the data in a TSDB appender. After reaching a *maxPerAppender* threshold depending on the available memory, the data is persisted on disk.

The way adapters are implemented is based on reflection, so that the developer just have to define (i) a data model as a struct with tags reporting the information for each metric to store (labels, metric name and metric type), (ii) the connector to the old monitoring system (e.g., a MySql Database), and (iii) a Go Routine that produces data (e.g., rows from a table) to be sent as slice of instances of the original data model in the Go channel (e.g., list of rows).

Finally, based on the characteristic of the available historical data and the available resources (CPU and Memory), the user can define the duration of a TSDB Block, the maximum number of chunks to store in the tree, the maximum number of metrics to keep in memory before actually storing them in the disk. This gives the user a very simple way to tune the back-filler and to perform the migration of data fastly.

As shown in Table 8.1, the Go Prometheus Backfiller demonstrated an improvement of 838% in average rapidity when performing 10 Gigabytes back-filling of Gauges stored in a MySql database, and made a better use of the available memory (16 GB vs 50 GB) on a LXC container hosted in a 2x Intel Xeon E5-2630 v4 @ 2.20 GHz server with 64 GB reserved memory and 20 reserved cores:

Another test was made by importing the Alibaba cluster trace from 43GB of pre-processed parquet files on a Core i7-5860K with lower resources than the first test: the results are shown in table 8.2. In this case, the producer Go routine is implemented to send in the channel at most 2000 chunks groups of 5 metrics with the same timestamp per-iteration. The code for this test is available as an example in the *go-prometheus-backfiller* repository.

Table 8.1: Comparing *promtool* and *go-prometheus-backfiller* on a 10GB MySql database

Data-set size	10 GB
Block duration	15 days
Max items per appender	100M metrics
Max chunks per BST	1k
Tables per second (go-prometheus-backfiller)	5.87 Tables/s
Tables per second (promtool + OpenMetrics conversion)	0.7 Tables/s
Memory (go-prometheus-backfiller)	50 GB
Memory (promtool)	16 GB

Table 8.2: Test parameters of back-filler for the Alibaba Cluster Trace v2018

Data-set size	43GB
Block duration	8 days
Max items per appender	100M metrics
Max chunks per BST	1k
Tables per second (go-prometheus-backfiller)	157.02 Chunks/s
Memory (promtool)	8GB

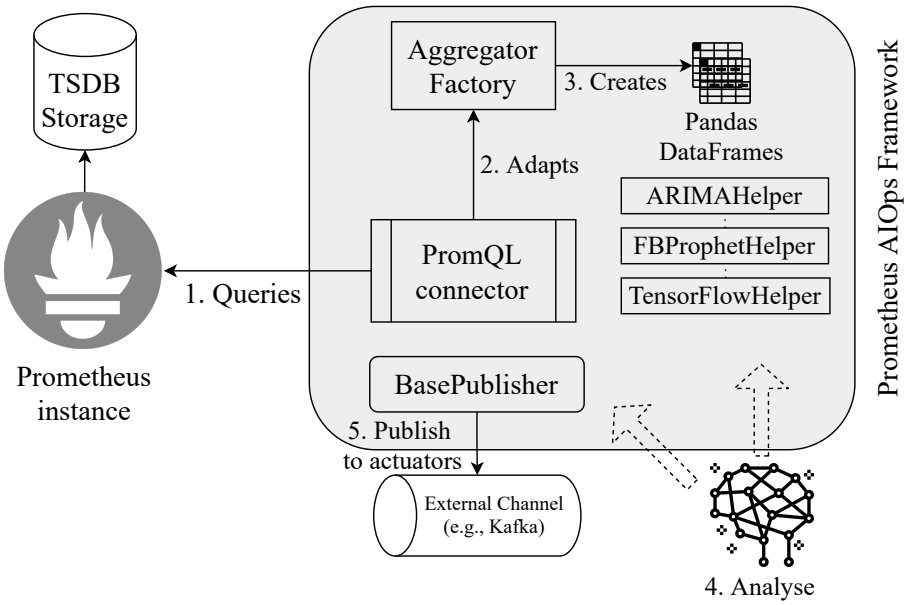

Figure 8.2: Architecture of the Prometheus AIOps framework

8.2 Enabling data-analysis on Ananke and Prometheus

Once the metrics are migrated from the old monitoring system, Prometheus will continue polling micro-services to collect new metrics.

Prometheus is already able to execute analysis on the available metrics and users can set recording rules and alerts to perform basic anomaly detection for the monitored applications.

Prometheus time series and its data analysis tools are useful in order to either visualise or operate on data by applying basic analytic operations (i.e., integration, linear regression, holt winters analysis): these operations are helpful for a user that wants to implement his AI-based algorithm for forecasting or anomaly detection purposes.

Most of the tools for data analysis and time-series forecasting are implemented in Python with the aid of libraries like Pandas, TensorFlow, StatTools, SciKit Learn, etc...

Usually, operators and researchers need to implement connectors to their data providers in order to build the in-memory DataFrames and models and run their algorithms. In production-grade environments, algorithms would be implemented as micro-services jobs that instruct the orchestrators to perform decisions support.

To this purpose, a second Python library is proposed: *Prometheus AIOps Framework*. It aims to support communication between Prometheus and the tools to perform analysis in *Ananke*.

The *Prometheus AIOps Framework* software architecture is depicted in fig. 8.2 and consist of the following modules:

- the *connector* module is responsible for the communication with the Prometheus APIs by providing a proxy for PromQL queries extending the official Prometheus API Client library: raw range queries, labels/metrics discovery, resampled and/or (on-prometheus) pre-processed time-series querying;

- the *aggregator* module is an adapter that convert any of the query results in ready-to-go Pandas DataFrames;

- the time-series *helpers* module implements basic operations on the DataFrames got from the *aggregator* to prepare data for usual strategies as ARIMA-flavoured models, Facebook Prophet [33] and RNNs/LSTM;

- the *publisher* module provides base facilities as classes to be extended: its purpose is allowing the publication of the analysis results over the network (e.g., Kafka messages in the case of *Ananke*)

Table 8.3: 1 minutes sampled Wordpress dataset description

Metric	Mean	Std	Min	Max
http_requests_20x	0.70	4.46	0	321
http_requests_30x	0.18	1.99	0	160
http_requests_40x	0.05	0.79	0	110
http_requests_50x	0.01	0.08	0	11
unique_conns	0.30	0.82	0	38
http_requests_GET	0.73	4.83	0	293
http_requests_POST	0.18	1.62	0	321

so that the orchestrator can decide the kind of operation to perform in the cluster;

- for debug and development purposes, the *debug* module enable integration with the IntelliJ remote debugging and scientific mode tools and with Jupyter.

The library also provides a Base Docker image ready to be used as a job for continuous feedback pipelines. With these facilities, a user can implement an *Ananke* analyser micro-service job that interact with Prometheus running the algorithms that finally push their results as commands for the PaaS cluster orchestrator.

8.3 Case study

Based on the architecture depicted in the previous section, this section shows, as a case study, the analysis of the traffic for a Nginx web server configured to be the edge proxy of a WordPress Blog. Forecast of the next day traffic (i.e., unique connections) is performed considering a high increase of it as an anomaly that has to trigger the scale-out of the systems at a specific time before the actual traffic peek occur.

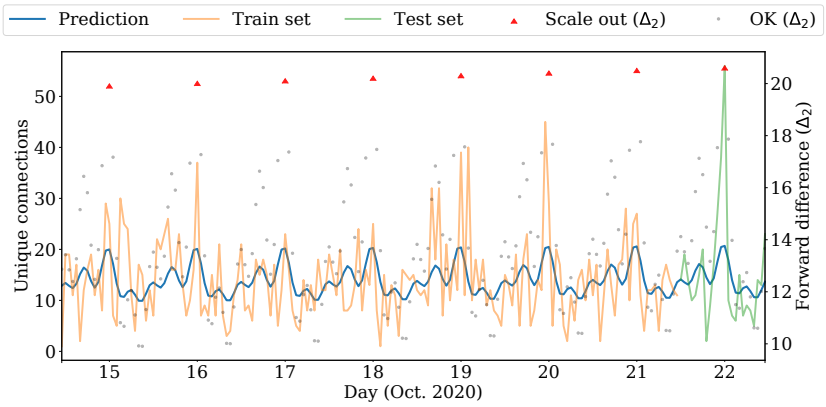

Figure 8.3: Prediction of the next day traffic peek as scale-out trigger

Table 8.4: 1 hour (re)sampled Wordpress dataset description

Metric	**Mean**	**Std**	**Min**	**Max**
http_requests_20x	42.18	88.05	0	2402
http_requests_30x	10.76	26.37	0	493
http_requests_40x	2.70	10.06	0	235
http_requests_50x	0.29	1.65	0	39
unique_conns	17.84	15.95	0	222
http_requests_GET	43.89	74.68	0	859
http_requests_POST	10.55	50.24	0	2373

Available data is stored in a generic nginx access log from October 2020 to March 2021.

Each row of the raw log represents a single request: data are imported the metrics into Prometheus as 1-minutes *gauges* samples providing map, filter and reduce functions to the Back-filling library presented in sec. 8.1. All data sample reports the cumulative sum describing the number of GET/POST requests and 20x/30x/40x/50x responses in the given time-window; moreover, for each minute, the unique connections are obtained

Table 8.5: RMSEs comparison between ARIMA, SARIMA, VARMA

Model	**RMSE**	**% vs Prophet**
ARIMA	11.25	129%
SARIMA	18.72	215%
VARMA	12.70	146%
Prophet	8.69	-

exploiting a filter based on the IPs from the "*X-Forwarded-For*" HTTP header available in the log to calculate the counter of unique connections.

The *Prometheus AIOps Framework* of sec. 8.2 is used to query Prometheus and to get chunks of the metrics on sequential time windows resampled with a window size of *30 minutes*, *1 hour*, and *2 hours*. Tables 8.3, 8.4 report the statistical description of the metrics with 1 minute and 1 hour sample rate. We compare the performance of ARIMA, SARIMA, VARMA [179] and Facebook Prophet [33] models to predict peek events in the unique connections to the end of informing the orchestrator and require a scale-out of the related micro-services in time before a peek actually occurs.

Orders for the ARIMA, SARIMA and VARMA models are chosen for each sequential training set automatically by using the *auto_ arima* facility from the Python library *pmdarima*.

Seasonality of SARIMA was set to 24 hours as the WordPress is mainly visited by users on the same time-zone. VARMA was used to test multi-variate forecasting with *unique connections* and *http_ requests_ GET* as features. Finally, Prophet is used with yearly seasonality disabled, and keeping just the weekly and daily seasonality enabled. In fact, a weekly seasonality seems reasonable for the kind of monitored application (e.g., readers of a blog could increase in non-working days).

We evaluate the models with their root mean squared error by using 168 samples for training (1 week hourly data) and 24 samples for validation (1 day ahead hourly prediction): as shown in table 8.5, RMSE for Prophet was the smallest: ARIMA and SARIMA performed, respectively, more than $1X$ and $2X$ RMSE with respect to the Prophet one.

Then, the output of Prophet forecast is employed to predict peek events and, when they arise, require the scale out of the micro-service.

In particular, these 24 samples are the input of a binary classifier that looks at the second finite forward discrete difference of the forecast and compare it with a threshold ϵ. To not scale twice or more times in the same time window also when scaling is not needed, we only scale if the second forward difference is greater than a threshold ϵ, and the first forward difference is less than ϵ.

Formally, given the forecast $f(t)$:

$$\mathrm{T}_{scale\_out} = \left\{ t \middle| \Delta_2[f](t) \geq \epsilon \wedge \Delta_1[f](t) < \epsilon \right\} \tag{8.3}$$

Fig. 8.3 shows the results of the prediction of a future day for the first week of the dataset: test data is represented by the green line while the orange line was used as training for the model. The red carets represent the events as classified by eq. 8.3.

The knowledge given by the set T_{scale_out} allows an agent to scale out the system in advance: the orchestrator can be set, as an example, to scale the micro-service (i.e., its *ReplicaSet*) 1 hour (or 30 minutes) before the peek is expected; moreover, the agent can run cross validation of the trend a few time slots before the scaling job is triggered so that false positives events could be detected.

When the traffic, finally, decreases, scale-in of the system can be triggered by already known horizontal auto-scalers looking at the cur-

rent resource usage of the system, i.e., horizontal pod auto-scaler in Kubernetes.

8.4 Conclusion

This chapter presented the design and implementation of the facilities to migrate old monitoring systems data of a cloud-native application to Prometheus instances of the *Ananke* framework and to enable integration of the time-series stored in Prometheus with standard tools usually available in the Pythonic fauna of AI libraries for data analysis. These facilities are available as Free Software in a Golang library (the Prometheus back-filler) and a Python library (the Prometheus AIOps Framework). Real executions of the back-filler show how it can easily allow bulk-import of time-series data from any format into TSDB and how it performs with respect to the current available official *promtool* in Prometheus.

The Prometheus AIOps Middleware is finally used to model the traffic of a WordPress application and predict the triggers to scale the system before the peeks actually occur. Initial results are shown to demonstrate how the Facebook Prophet model combined with a binary classifier can predict those peeks efficiently to perform scaling of the cluster.

The next chapter will continue this work by employing a deeper AI pipeline that will make use of a reinforcement learning agent.

It will decide whether scale out or reconfiguration of the underlying network should be performed when SLIs are predicted to violate the SLA between a customer and their provider.

Chapter 9

NAPA: to scale or not to scale

«Day by day, what you choose, what you think and what you do is who you become.»

Heraclitus

The work done in previous chapter allowed getting (i) the model and architecture to enable fine-grained monitoring and orchestration of micro-service based applications, *Ananke*, and (ii) the framework to ingest black-box monitoring data and analysing them. This chapter presents NAPA, Network Adaptiveness Pod Auto-scaler.

It is based in the black-box monitoring tools of Ananke, depicted in the previous chapter.

With respect to the previous case study in 8.3, NAPA is proposed as a generalized strategy that leverages a reinforcement learning agent to support decisions of an orchestrator when SLOs are predicted to be violated, deciding whether the best action to perform is either to scale-out a

deployment Kubernetes object or re-configure the underlying network for better handling the network traffic with the already allocated resources.

As Ananke relies on OpenTracing APIs [176] to define monitored metrics data structures, in this work the OpenSLO initiative [23] is employed for the definition of service-level objectives through Kubernetes Custom Resource Definitions (CRDs). Finally, the workers network is considered to be software-defined and based on Flannel [180]. jFlowlight [106–108], and the *A4SDN* algorithm are employed to properly set up network rules and optimise inter-pod and inter-worker communication.

In order to allow handling of the SLOs with OpenSLO, a modification to the graphs in eq. (8.1) and (8.2) has to be considered to add predicates into the Ananke's data model.

In particular, a set $\mathcal{P}_A$ for the definition of the predicates stated in SLAs is added:

$$\mathcal{G}_{C,t}(\mathcal{M}_w, \mathcal{M}_w^2, P_\tau^w(w, k_w), P_\tau^l(v, w, k_l)) \tag{9.1}$$

$$\mathcal{G}_{A,t}(\mathcal{V}_A, \mathcal{V}_A^2, P_\tau^A(w, k_w), P_\tau^l(v, w, k_l)) \tag{9.2}$$

where:

- $\mathcal{M}_w$ is the set of workers VMs;
- $\mathcal{V}_A$ is the set of micro-services (i.e., *Deployments, Statefulset, ...*);
- $P_t^w \colon \mathcal{M}_w \times \mathcal{K}_w \to \mathbb{U}$ is a function that maps a key in the set $\mathcal{K}_w$, for a given vertex $w \in \mathcal{M}_w$, to the value of the property (e.g., CPU/Mem usage);
- $P_t^l|_{\mathcal{M}} \colon \mathcal{M}_w^2 \times \mathcal{K}_l \to \mathbb{U}$ is a function that maps a key from the set $\mathcal{K}_l$, for the given edge between vertices $v \neq w \in \mathcal{M}_w$, to the value of the property (e.g., network bandwidth).

- $P_t^A : \mathcal{V}_A \times \mathcal{K}_A \rightarrow \mathbb{U}$ is a function that maps a key in the set $\mathcal{K}_A$, for a given vertex $V \in \mathcal{V}_A$, to the value of the property (e.g., CPU/Mem usage for the specific pod);

- $P_t^l|_A : \mathcal{V}_A^2 \times \mathcal{K}_l \rightarrow \mathbb{U}$ is a function that maps a key from the set $\mathcal{K}_l$, for the given edge between vertices $v \neq w \in \mathcal{V}_A$, to the value of the property (e.g., network usage of a pod);

- $\mathcal{P}_A$ is a set of 4-tuples representing the SLOs. Each 4-tuple consists of (i) a node $v \in \mathcal{V}_A$ (ii) an operator in the set $\{\leq, \geq\}$, (iii) a key $k \in \mathcal{K}_l \cup \mathcal{K}_w \cup \mathcal{K}_A$, and (iv) a target value depending on the kind of metric to which the predicate refers.

9.1 Replication management in Kubernetes

Kubernetes simplifies the deployment, management, and execution of containers on distributed computing resources.

In K8S, users define resources by leveraging a declarative approach that emphasizes one of the most important features offered to deploy applications as a code: separation of concerns for resources allocation and management, based on the label-selectors pairing in a master-worker architecture. The master with its services is in charge of preserving, at run-time, the desired state of the cluster by managing resources deployed at the workers. A worker node is a physical or virtual machine that offers its computational capability for executing pods in a distributed manner. Pods are the smallest deployment unit in K8S consisting of one or more containers deployed at the same worker, and sharing the same *namespace*; this means that containers running on the same pod communicate with each other either through the loopback network interface or over inter-process communication channels.

For each container, *requests* and *limits* identify respectively resources to be allocated for a given pod during the placement, and the limit of resources after which a pod can be evicted from the worker (and rescheduled by kube-scheduler). In the lower-level system, limits and requests lay to define *cgroups* properties related to the containers.

Resource requirements, in this chapter, will be considered as the sum of requirements by resource type for each container of a given pod.

K8S can run multiple replicas of a pod according to the *ReplicaSet* capability. It ensures that a fixed number of pods are up and running for a given higher-level owner object. *Deployments*, as an example, are higher-lever abstractions that encapsulate different historical versions of *ReplicaSets* for the same kind of pods in order to allow definition of update policies (i.e., Rolling Updates, Recreate, etc. . .).

9.2 Horizontal auto-scaling

K8S provides basic auto-scaling facilities for pods by the *HorizontalPodAutoscaler* (HPA) capability.

This basic functionality consists of a scaler that uses current resource usage to trigger scale-out or scale-in of a *ReplicaSet* when one or more threshold based conditions are satisfied.

Thresholds can be defined for each default resource by defining a type as Utilization (percentage) or AverageValue (according to the measure unit of the resource).

Default available resources are CPU and Memory, but custom (application specific) definition of resources is also possible, enabling the use of the punctual *Value* target type.

The scaling policy of the auto scalers can be configured to define stabilization windows, percentage of pods to scale and selection policy

to force the HPA controller to decide how many pods to schedule (or destroy) by iteration [181].

In particular, the HPA controller calculates the desired number of replicas at time t as in the following:

$$r_t = \left\lceil r_{t-1} \cdot \frac{SLI}{SLI^{target}} \right\rceil \tag{9.3}$$

Finally, when conditions for scaling are met Kubernetes HPA can be configured to either scaling by a fixed number of pods or by a percentage of pods. A stabilization time window is exploited to wait for load to be balanced across new replicas.

9.3 Flannel

Differently from the default SDN containers network driver, Flannel does not employ two SDNs to route traffic between containers in different hosts: as a consequences, it allows for fine-grained management of the data-path between hosted pods and workers, avoiding double encapsulation of packets whenever the workers network itself is also an SDN.Flannel provides a layer 3 IPv4 network between multiple nodes in a cluster.

With Flannel, each worker within the cluster runs a daemon, *flanneld*, that is responsible for (i) allocating a unique subnet on each host, (ii) distributing IP addresses in the host-specific subnet to the containers, and (iii) mapping routes for inter-container communication, even if on different hosts. The daemon publish information about pods-host networking to the *etcd* store so to distribute routing knowledge to the other hosts' daemon instances. Inter-worker communication is backed by the *host-gw* [1] flannel mode based on direct layer-2 connectivity, which

[1] https://github.com/flannel-io/flannel/blob/master/Documentation/backends.md

makes it compatible with the *spine and leaf* employed in the scenario of the proposed work.

9.4 Network-aware scaling or network adaptiveness

Kubernetes designers continuously improve HPA objects.

The description given in the previous section is based on the *v2beta2* Open API specification available since Kubernetes v1.18.

HPA configuration can be tricky and can also be unable to intercept scaling issues just in time when peek events occur. As a consequence, the service provider can violate SLOs stated in the SLAs during these events.

The work exposed in the previous chapter, compared the performance of ARIMA, SARIMA, VARMA [179] and Facebook Prophet [33] algorithms to predict peek events from unique connections metrics of a micro-service to the end of scale-out automatically the related micro-services in time before a peek actually occurs.

The models have been evaluated with their root mean squared error (RMSE) by using 168 samples for training (1 week hourly data) and 24 samples for validation (1 day ahead hourly prediction): RMSE for Prophet was the smallest. ARIMA and SARIMA performed, respectively, more than $1x$ and $2x$ RMSE with respect to the Prophet one.

Then, the output of the Prophet forecast has been used to predict peek events through a binary differential classifier.

These peeks of traffic lead scale out of the micro-service.

However, in the HPA object and in Kubernetes in general, there is no way to use network-aware metrics to control an application: the default scheduler does not involve any network-aware parameter when deciding

which of the workers has to host a pod and, in order to make use of network policies within an HPA, a user should define custom resources and application-specific metrics.

The last versions of Kubernetes APIs implements NetworkAware-Policies that can only suggest the scheduler how to place pods in the workers or, conversely, set mandatory Affinity/Anti-Affinity rules to re-schedule pods after their first deployment to a suitable (policy enabled) worker [44].

Scaling out stateless micro-service is considered today a de-facto standard: however, there can be environments for which scaling pods can be more expensive than the actual performance gain the application would get in the state at which HPA triggers the scale-out operation.

As an example, let's suppose to have a pod that needs a lot of memory/CPU resource to boot up: a custom network-aware HPA would scale-out the ReplicaSets when network traffic to the Pod arises to a non-acceptable level with respect to its SLA (e.g., for which the throughput starts to decrease, or the response time becomes too high). Scaling the pod would lead to a distribution of the load that can fix the issue and make the system back to the desired state as defined in a SLA.

However, the pod would just require additional network resources, more bandwidth or another packets queue with a good reservation policy to guarantee QoS. In such a case, scaling out the ReplicaSet led to waste of resources (needed for the boot-up and the background threads) and could have been avoided if an operator would adapt the lower-level software-defined network to improve communication performance of the pod or the underlying worker.

It can be the example of a Java Spring application that make use of a network persistent volume remotely hosted in another worker: in a *spine and leaf* architecture, reconfiguration of the routes or just the

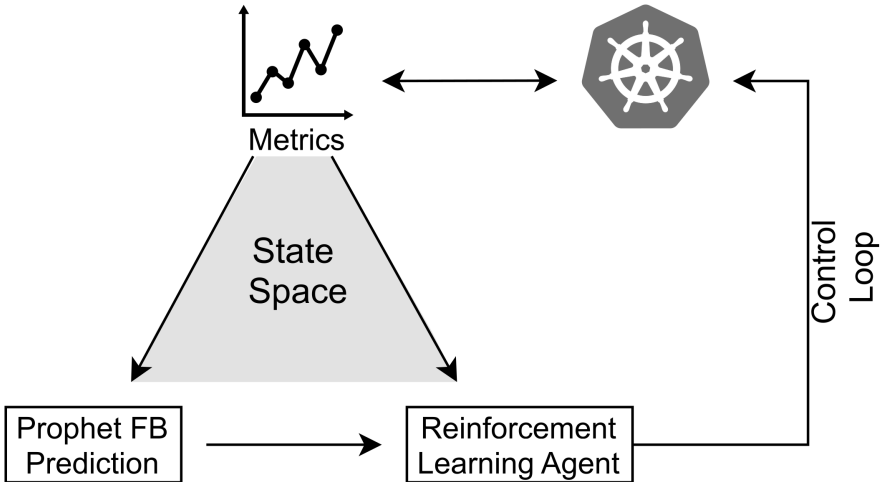

Figure 9.1: RL Pipeline

reservation of one more communication channel for that only worker (and pod) would fix just in time traffic peeks issues like huge remote I/O operations in a GlusterFS volume.

This work proposes a reinforcement learning strategy which combines a time series forecasting model together with current metrics of the monitored application as the state space for a Q-Learning [182] agent aiming to minimise risk of violating SLOs stated in the SLA of a given application.

9.4.1 *NAPA*: Network-Aware Pod Autoscaler

Fig. 9.1, basically exposes this machine-learning pipeline. NAPA will consider the micro-services of a single application A and will extend the strategy to multiple applications sharing the same environment at the last.

Reinforcement Learning (RL) collects trial-and-error steps by which an agent can learn how to make good decisions through a sequence of interactions with the environment.

At any decision step t, the agent determines the system state $\boldsymbol{s}(t)$ using a subset of the metrics coming from $P_\tau^w, P_\tau^l, P_\tau^A, P_\tau^{l,A}$ in def. 8, calculates the immediate cost $c(t)$ by the means of the SLOs defined through the 4-tuples $\mathcal{P}_A$, and updates the expected long-term cost (i.e., Q-function $Q(s,a)$).

Actions of the agent for any micro-service $v \in \mathcal{V}_A$ lay in the set $\mathcal{A} = \{NOP, ScaleOut, AdaptNet\}$, where NOP is the null operation (no perturbation is needed to optimise the cluster state), $ScaleOut$ adds R replicas of the pod to the ReplicaSet and $AdaptNet$ trigger reconfiguration of the network QoS for the hosting worker in the leaf-spine network up to the Pod virtual interface.

Exploration vs Exploitation

One of the main challenges is to find a good trade-off between the exploration and exploitation phases. To minimise the obtained cost, a RL agent must prefer actions that it have shown to be effective in the past (exploitation). However, to discover such actions, it has to explore new actions (exploration).

Here, the simple $\epsilon - greedy$ policy for which a threshold ϵ is used to decide, at each iteration, if make use of exploitation (i.e., trigger a random action with probability $1 - \epsilon$) or executes the action with the lowest long-term cost (exploitation):

$$a(t) = \arg\min_{a' \in \mathcal{A}} Q(\boldsymbol{s}_t, a') \tag{9.4}$$

Long-term cost

The proposed strategy aims to minimise the Q-Value on each iteration of the agent life cycle.

As a consequence, at the end of each interaction, the long-term cost is calculated by:

$$\begin{aligned} Q_{t+1}(\boldsymbol{s}_t, a) \leftarrow (1-\alpha) \cdot Q_t(\boldsymbol{s}_t, a) \; + \\ \alpha \cdot [c_t(\boldsymbol{s}_t, a) \; + \gamma \min_{a' \in \mathcal{A}} Q_t(\boldsymbol{s}_{t+1}, a')] \end{aligned} \tag{9.5}$$

where:

- α is the learning rate;
- γ is the discount factor;
- c_t represents the risk factor at time t (when the action is executed);
- $\boldsymbol{s}_{t+1}$ is the next state (after the execution of the action a).

Immediate cost (risk factor)

Derived from the set $\mathcal{P}_A$, a generic SLO SLO_i^A for a micro-service $v \in \mathcal{V}_A$ can be described as the predicate $SLI_i^A \gtrless SLI_i^{A,target}$.

In a generic SLA, we can partition its SLOs in $n_{\geq}^A$ - for any $SLI_i^A \geq SLI_i^{A,target}$, and $n_{\leq}^A$ - for any $SLI_i^A \leq SLI_i^{A,target}$ - predicates.

As previously anticipated, $\mathcal{P}_A$ is the set of SLOs related to application A. It's trivial to say that cardinality of $\mathcal{P}_A$ is $|\mathcal{P}_A| = n_{\leq} + n_{\geq}$.

NAPA has to allow decision making based on the risk of violating one or more SLOs. Violating an SLO means to get, at a specific time, SLIs that do not respect the predicates in $\mathcal{P}_A$.

All the $|\mathcal{P}_A|$ SLOs identify a surface in a $|\mathcal{P}_A|$-dimension space $\mathbb{S}$. For a given time t, the SLIs of the application defines a vector $\boldsymbol{s} \in \mathbb{S}$: violating SLA means the state of the application (its SLIs) identify a point outside the surface defined by $\mathcal{P}_A$.

Therefore, in the risk factor calculation, a term exploiting the notion of distance between target SLIs and observed values has to be introduced: to this aim, a modified version of the Canberra distance [183, 184], that ranges in [-1,1[, has been exploited:

$$d(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{n \cdot W} \cdot \sum_{i=1}^{n} w_i \cdot \frac{p_i - q_i}{p_i + q_i} \tag{9.6}$$

where w_i are weights and W is the sum of all the weights. The notion of distance is actually misused since the one in eq. (9.6) is non-symmetric, can be negative and do not satisfy the triangle inequality. However, it represents an index depicting the risk of violating the SLA, assuming SLIs and SLOs thresholds are always non-negative. In fact, in the 1-dimensional case, $d(p, q)$ would lead to a value in $[-1, 1[$, with 1 being the asymptote and -1 when $p \equiv 0$, see fig. 9.2.

With eq. (9.6), the normalized risk of violating the SLA is defined as:

$$\begin{aligned} c_{SLA}(\boldsymbol{s}, \mathcal{P}_A) = \frac{1}{|\mathcal{P}_A| \cdot W} \Bigg[& \sum_{j=1}^{n_{\leq}^{A}} w_j^A \cdot \frac{SLI_j^A - SLI_j^{A,target}}{SLI_j^A - SLI_j^{A,target}} + \\ & \sum_{j=1+n_{\leq}^{A}}^{1+n_{\leq}^{A}+n_{\geq}^{A}} w_j^A \cdot \frac{SLI_j^{A,target} - SLI_j^A}{SLI_j^{A,target} + SLI_j^A} \Bigg] \end{aligned} \tag{9.7}$$

Finally, a term for the resources booking and another one that allow to avoid unfeasible actions should be introduced. Given L resource types (e.g., CPU, memory, storage...), let's define:

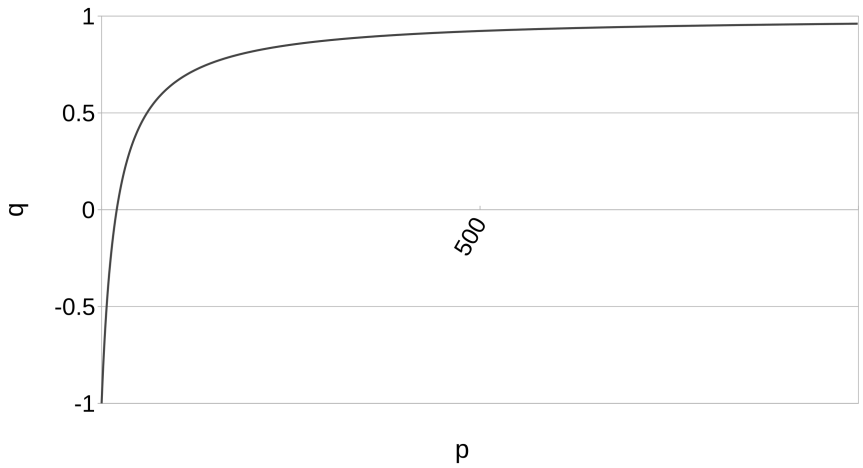

Figure 9.2: Modified Canberra 1D distance d(p, q), with fixed $q > 0$, for $p \geq 0$

1. $\boldsymbol{C}$: $P_t^A \rightarrow \mathbb{R}^L$ the L-dimensional vector representing the limits of resources of the application A (i.e., the limits of the application multiplied by the number of replicas);

2. $\boldsymbol{U}_l$: $P_t^A \rightarrow \mathbb{R}^L$ the L-dimensional vector representing the resources occupied by the application A (i.e., the sum of utilized resources by each pod of the application);

3. $C_{net}(P_t^l|_A)$ the capacity of the network for the application;

4. $U_{net}(P_t^l|_{\mathcal{M}})$ the utilized network.

We define a piecewise function for the resource booking as:

$$c_{res}(s,a) = \begin{cases} 0 & \text{if } a = NOP \\ d(\boldsymbol{U},\boldsymbol{C}) & \text{if } a = ScaleOut \wedge \\ & \quad d(\boldsymbol{U},\boldsymbol{C}) > 0 \\ d(U_{net},C_{net}) & \text{if } a = AdaptNet \wedge \\ & \quad d(U_{net},C_{net}) > 0 \end{cases} \tag{9.8}$$

Given eq. (9.7) and eq. (9.8), the total immediate cost considered is the following:

$$\begin{aligned} c_t(s,a) = & h_1 \cdot c_{SLA}(\boldsymbol{s},\mathcal{P}) + \\ & h_2 \cdot c_{SLA}(\boldsymbol{s}_{pred},\mathcal{P}) + \\ & h_3 \cdot c_{res}(\boldsymbol{s},a) \end{aligned} \tag{9.9}$$

where h_1, h_2, h_3 are weights and s_{pred} in the second term comes from the output of the FBProphet prediction.

The action chosen by the agent can eventually be unfeasible (e.g., the available resources in the cluster are not enough to host a new replica for a given $v \in \mathcal{V}_A$). Despite placement is generally a NP-Complete problem [4], especially in the case of a cluster consisting of multiple heterogeneous workers, in this scenario, an action would lead either to the deployment of a new replica for a given application or to the network re-configuration: feasibility in both cases can be verified more easily as the Kubernetes scheduler already does when deciding for the placement of a pod in the scheduler queue: just check whether the filter step of kube-scheduler produces an empty set of workers by looking at the available resources and at the required resources. In this way, unschedulable pods will never be submitted to the kube-scheduler facilities.

If an unfeasible action is chosen, the agent changes the action to the NOP one.

9.4.2 Extension to multiple applications and implementation

Since a Kubernetes cluster is usually exploited as a shared environment for multiple customers or applications, the agent has to leverage more actions than the 3 ones described previously. In particular, let's consider a set of applications $\mathcal{N}$ consisting of a set of pods $\mathcal{C}$.

Extension to the case of multiple applications to handle leads to the definition of a dynamic environment both in terms of actions and states [185, 186].

In particular, the dimension of the space of states would increase or decrease, respectively, in the case of the arrival of a new application or the deletion of an old one. The space of states will depend on the SLOs required by the SLAs of each application.The calculation of the immediate cost as in eq. (9.9) can be obtained still using the modified canberra distance, as before, but across all the running applications' SLOs.

The actions space is more straightforward: we can redefine the set of actions A by:

$$\mathcal{A} = \{NOP\} \cup \bigcup_{A \in \mathcal{C}} \{ScaleOut_A, AdaptNet_A\}$$

A general pseudocode of the algorithm that enables *NAPA* is reported in alg. 9.1.

Note that the states of the cluster have to be in a form such that they can work as an index for the Q matrix.

In order to allow this kind of indexing, states that lay in a continuous space (i.e., metrics in the realm of real numbers $\mathbb{R}$) are passed through a quantization step, as exposed in line 5 of algorithm 9.1.

9.5 Architecture and workflow

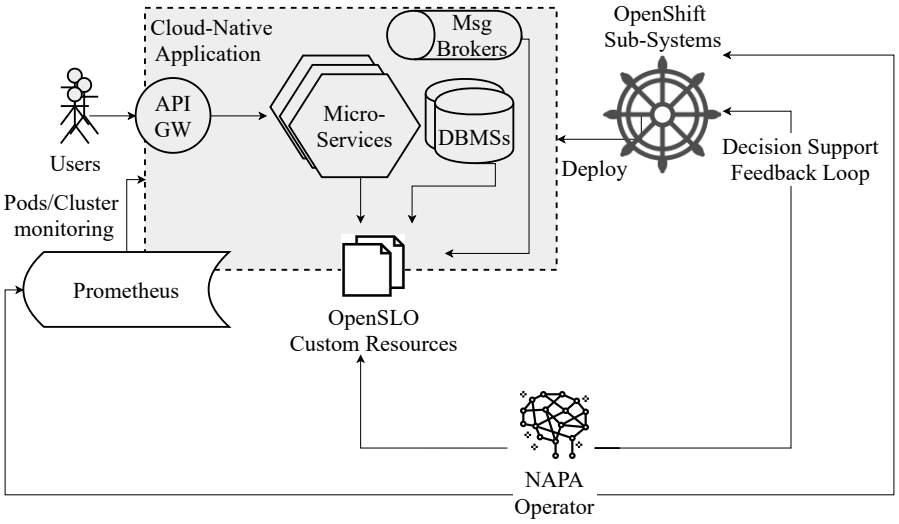

Figure 9.3: *Ananke* architecture

Fig. 9.3 highlights the Ananke's distributed architecture which is spread between edge and cloud clusters, with a specific focus to the *NAPA* workflow requirements.

Here, the main different components relate to (i) the OpenSLO CRDs, and (ii) the *NAPA* operator responsible for processing the time series of applications' behaviour, and the execution of the agent that enables the decision support feedback loop. It represents an online analyser in the general architecture from fig. 7.5.

The elements needed by the architecture to run this approach are:

- monitoring facilities to expose SLIs related to the pods;
- a standard way to define the SLOs;
- a job executing the ML pipeline depicted in the previous section;
- finally, the control loop controller.

9.5.1 Pods and cluster monitoring

Monitoring of the whole system has to consider (i) workers resource utilization (CPU, memory, network...), (ii) general pods resource utilization by replicas and, finally, (iii) application-specific key performance metrics like average response time, end-to-end latency, throughput, etc..., according to the SLOs required.

Both OpenShift and vanilla Kubernetes offer ways to continuously monitor workers and hosted pods. One of the most used enabling technologies is Prometheus [187]: it comes out of the box with the free version of OpenShift - OKD - and well fits the first two requirements of the monitoring capabilities neeeded by the herein proposed strategy.

The third requirement, application-specific metrics, can currently be achieved in different ways: Google Kubernetes Engine make use of annotations on the K8S objects definitions; differently, OpenShift make use of the the Custom Resource Definition (CRD) *ServiceMonitor* to define targets for the managed prometheus instances.

Both these solutions are compliant with the architecture of *Ananke* and can be exploited by the *NAPA* facilities.

Finally, when migrating an application to a cluster managed by *Ananke* and *NAPA* has to be trained, the application providers and the resource provider may need to ingest historical data previously collected

by the previous monitoring system in order to allow, as an example, faster training of the analysers.

The back-filler from the previous chapter can be exploited for this case.

With facilities from previous chapters and considerations above, a user can implement an *Ananke* analyser micro-service job that interact with Prometheus, running the algorithms of the *NAPA* agent, and finally letting the controller able to orchestrate the cluster through the operator pattern and the cluster's infrastructure-as-a-code APIs.

Algorithm 9.1 Action selection function

```
s_prv, s_cur: Tuple
Q: Matrix[len(States)][len(Actions)]
last_action: Action
while True:
    s_cur = quantize(get_current_state())
    if not s_prv: # initialization phase
        initializeQwithRandomValues()
    else:
        # Updates Q matrix on next iteration
        # after action execution
        Q[s_prv][last_action] = (1 - alfa) * \
        Q[s_prv][last_action] + alfa * \
        (cost(s_prv, action) + gamma * \
        get_min_cost(s_cur))
    if (1 - epsilon) < randomFloat():
        # exploration
        last_action = randomAction()
    else:
        # exploitation
        last_action = get_min_cost_action(s_cur)
    if not check_feasibility(last_action):
        last_action = NOP
    execute(last_action)
    # Save previous state
    s_prv = s_cur
    # Wait for a stabilization window
    # before taking a new action and
    # update the Q matrix
    sleep(T)
```

Listing 9.2 Service monitoring in GKE

```
apiVersion: v1
kind: Service
metadata:
 annotations:
   prometheus.io/scrape: 'true'
   prometheus.io/path: '/metrics'
# ...
```

Listing 9.3 ServiceMonitor CRD on OpenShift

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
    labels:
        k8s-app: prometheus-example-monitor
    name: prometheus-example-monitor
    namespace: ns1
spec:
    endpoints:
        - interval: 30s
          port: web
          scheme: http
    selector:
        matchLabels:
            app: prometheus-example-app
```

9.5.2 Defining the SLOs

SLOs of a given pod managed by its *ReplicaSet* or *Deployment*, need to be discoverable by the system level services running *NAPA*. Kubernetes and its APIs are designed to allow extensions of the system following best practices and patterns typical of the software engineering applied to the field of operations management: separation of concerns, encapsulation of operational knowledge, decoupling. In order to provide encapsulation of operational knowledge and to have vendor-agnostic design of *NAPA*, the list of SLOs that the RP has to guarantee for a given pod, have been defined based on the OpenSLO initiative's proposal.

OpenSLO [23] is a service level objective language based on YAML. It allows for definition of targets specification that comes from an SLA in a declarative way.OpenSLO is designed to be implementation neutral so that SLOs can be shared in well-defined formats between multiple environment, e.g., on heterogeneous multi-cloud.

OpenSLO defines a vendor-agnostic interface description language to define and track SLOs, compliant with the Kubernetes OpenAPI specification and flexible enough to be extended in other platforms.

The specifications provide two main kind of Object Types: *SLO* and *Service*.

A *SLO* represents a single SLO, with the given indicator and data source configuration, timing information and, finally, the objective itself. Data source can be described providing the name of the service that provides the observed metric (i.e., Prometheus) and the query to get the values for the objectives functions. Listing 9.4 reports specification of a partial OpenSLO custom resource that represents the minimum knowledge information to provide for *NAPA*.

A service represents a group of SLOs object. Binding of the SLOs is done through the *service* key of each object: therefore, the Service

object of the *openslo* API group has to be specified only once together with the definition of the related deployment. Binding of the Service to the observed object can be achieved with the standard label-selector pattern.

CRDs defined to enable encapsulation of SLOs knowledge for the cluster has to be handled by the proper controllers: the operator pattern can be useful to implement both the ML pipeline job, and the privileged pods to handle the control loop, i.e., the orchestration of Kubernetes resources, and the workers networking configuration.

9.5.3 *NAPA* Operator

A Kubernetes operator is an application-specific controller that extends the functionality of the Kubernetes APIs to create, configure, and manage instances of complex applications on behalf of a Kubernetes user.

K8S internal controllers implement control loops that repeatedly compare the desired state of the cluster to its current state. If the cluster's current state doesn't match the desired state, then the controller takes actions to fix the problem.

An operator can use custom resources (CR) to manage applications and their components. By using the CRD defined in the previous sections by the OpenSLO, the Kubernetes cluster get high-level configuration and settings for the managed applications. The Kubernetes operator translates the high-level directives into the low level actions, based on the logic embedded within the operator's logic.

We add the *NAPA* CRD to the ones provided by OpenSLO as the initial point of access that *NAPA* has to use to control either the scaling of applications, or the configuration of underlying network. The *NAPA* CRD, defined in listing 9.6, is to allow separation of concerns between

OpenSLO objects that could also be used by other operators in the cluster.

By the use of the *NAPA* CRD the operator has to:

- Get the NAPA CRDs;
- Get all the related SLOs needed as a state of the application;
- Implements the connectors to the horizontal scaling APIs of Kubernetes, and the interfaces to communicate with a privileged *DaemonSet* running on the workers to communicate decisions about configuration of the network executed through standard tools as kernel system-calls, iptables/tc APIs...
- Periodically, the operator has to update the state of the whole cluster and their applications by using the SLIs related to the enabled SLOs and the low-level metrics of the cluster itself; Then, the operator executes the algorithm in section 9.4 to decide the action that minimises the long-term cost of the cluster.

Listing 9.4 Definition of an SLO in OpenSLO

```
apiVersion: openslo/v1alpha
kind: SLO
metadata:
  name: string
  displayName: string # optional
spec:
  description: string # optional
  service: [service name] # name of the service to
      associate this SLO with
  indicator: # represents the Service Level Indicator
      (SLI)
    thresholdMetric: # represents the metric used to
        inform the Service Level Object in the
        objectives stanza
      source: string # data source for the metric
      queryType: string # a name for the type of query
          to run on the data source
      query: string # the query to run to return the
          metric
  objectives:
    - displayName: string # optional
      op: lte | gte | lt | gt # conditional operator
          used to compare the SLI against the value.
          Only needed when using a thresholdMetric
      target: numeric # budget target for given
          objective of the SLO
```

Listing 9.5 OpenSLO Service: grouping SLOs

```
apiVersion: openslo/v1alpha
kind: Service
metadata:
  name: string
  displayName: string
spec:
  description: string
  selector:
    matchLabels:
      app: monitored-deployment
```

Listing 9.6 NAPA custom resource definition

```
apiVersion: napa/v1alpha
kind: AgentSubscription
metadata:
    name: string
    displayName: string
spec:
    description: string
    openSLOService: string # name of the openslo/
        v1alpha1 service object
```

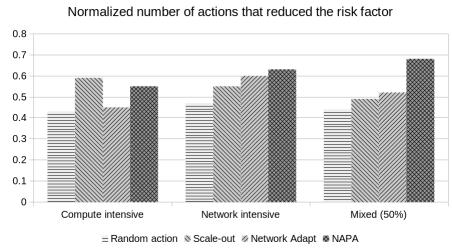

Figure 9.4: Number of actions that reduced the risk factor

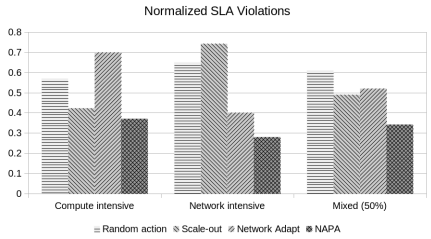

Figure 9.5: Number of SLAs violations for different test scenarios

9.6 Numerical results

Based on the algorithm presented in section 9.4, numerical simulations are presented to validate the proposed approach. In particular, the efficacy of *NAPA* is evaluated by the mean of a simulated environment in which multiple applications compete each other to gain resources in their execution environment, using a homogeneous space of states between the different applications.

Three kind of execution workloads running in the cluster are considered:

- computing-intensive applications for which an increase in the load (e.g., in terms of number of requests) implies the growing of CPU and memory usage; it is the case of micro-services that implement scientific jobs that involve big computation to complete their tasks [188];

- network-intensive applications for which network communication plays an important role to get their tasks completed, like distributed graph processing systems [2] or storage intensive applications based on distributed network attached volumes [189]

- a mixed environment consisting of 50% compute-intensive applications and 50% network-intensive applications.

The requests for these kinds of applications are considered independent events so that one can model them by using the Poisson distribution. For each simulation, 1000 iterations are executed and for each one of them, the number of correct decisions and the number of SLAs violation are evaluated.

These behavior are compared with the following four different agents/-controllers:

- Random agent: it chooses a random action from the set $\mathcal{A}$ for each iteration;
- Scale-out agent: it chooses between scale-out and the *NOP* action based on the risk of violating the SLA;
- Network re-configuration agent: it chooses between the network re-configuration and the *NOP* action based on the risk of violating the SLA;
- *NAPA*: the reinforcement learning agent proposed in this paper.

1000 iterations is because in simulations, with the learning rate parameter of the *NAPA* operator set to $\alpha = 0.2$, it got around 250 iterations to be trained enough to have consistent decisions.

The execution environment consists of homogeneous machines that can run at most 110 pods, as default in K8S.

9.6.1 Number of correct actions

Fig. 9.4 shows the normalized number of correct actions that each agent performed in the workloads defined above, i.e., the actions that result

in the reduction of the immediate risk factor. As it will be confirmed by the number of SLAs violations discussed in the next section, these results well expose as executing an action reducing the immediate risk does not automatically lead to the avoidance of SLAs violations.

In fact, reducing the immediate risk should not be considered the objective of a good SLA-aware operator. Objective is to avoid SLAs violations and reducing more times the risk factor of an application can, in the other side, leads to the violation of the SLAs for others applications interfering with the "optimised" one.

This is going to be confirmed in the following by looking at the results from fig. 9.5.

As expected, all the agents report better decisions than the random one, for the network-intensive and mixed workloads. Only the compute-intensive workload seems to be better handled, in terms of actions reducing the immediate risk factor, by the scale-out agent with respect to *NAPA*.

Network-intensive applications get ~$1.1x$ more reductions of the risk factor by the network re-configuration agent with respect to the scale-out agent, while computing-intensive workloads get ~$1.5x$ more reductions by the scale-out agent with respect to the network re-configuration agent. One can argue that these two agents expose better handling of the only network-intensive workloads for an agent managing the trade-off between the scale-out and the network re-configuration actions. *NAPA*, apparently, gives ~$0.84x$ good decisions in the case of compute-intensive workloads, and ~1.05 in the case of network-intensive, respectively compared to the scale-out and to the network re-configuration agent. Instead, in the mixed scenarios *NAPA* gains around $1.3x$ better decisions with respect to both the other two agents.

9.6.2 Number of SLA violations

Fig. 9.5 reports the number of SLAs violations normalized to the number of iterations performed by the agents in the three different scenarios depicted above.

As expected, the random action agent, reports a number of normalized SLA violations due to the actions chosen in the range between ~55% and ~60%, while the network-reconfiguration and the scale-out agents seem to expose a dual-like behavior for the cases of compute-intensive and network-intensive applications.

In particular, network-intensive applications are better handled by the network re-configuration agent than by the scale-out one, as well as the compute-intensive applications are better handled by the scale-out agent.

This was expected: the network reconfiguration agent directly affects network key parameters such as delay, jitter, packet error rate, and its effect is amplified in particular for network-intensive applications. In the same way, compute-intensive applications will have benefits from the increase of CPU/Memory resources allocation given by the scale-out operation.

In the homogeneous scenarios, the performance of the *NAPA* agent are ~184% better than those of the network re-configuration agent, while they are slightly better (~105%) in the computing-intensive scenario. Similarly, in the network-intensive scenario, *NAPA* performs ~240% better than the scale-out agent, and ~142% better than the network re-configuration agent. These results confirm statements of the previous sections about the impact of network on modern distributed CNAs.

Finally, the mixed scenario reports, for *NAPA*, around 157% less SLA violations with respect to the scale-out agent, and 141% with respect to the network re-configuration one, confirming how *NAPA* is able to handle

the optimisation of SLAs guarantees on scenarios of heterogeneous CNAs deployed on SDN-enabled clouds.

9.7 Conclusion

This chapter presented *NAPA*, an SLA-aware strategy for micro-service applications deployed on clouds that employ software-defined architectures for both the micro-services virtual network and the physical hosts communication.

This strategy aims to optimise both the network and the number of micro-services replicas executing in the cluster at runtime.

Novelty of *NAPA* is to combine (i) the forecasting techniques based on the Facebook Prophet model able to predict not only the behavior of time-series related to the cluster status, but also events such as peeks in requests per seconds, network traffic, resource consumptions. . . , and (ii) a Q-Learning agent that keeps estimation of the long term risk of violating SLOs stated in the SLAs of the hosted applications in order to perform the optimal decision and ask the cluster orchestrator to execute an action reducing this long-term risk.

Numerical simulations showed how *NAPA* is able to conduct a better management of the whole, cooperating, system, compared to naive agents that executes random actions or just one of them based on metrics thresholds typical of clouds environments.

Part IV

Conclusions and Future Work

Chapter 10

Final considerations

Fog/edge computing, function as a service, and programmable infrastructures, like software-defined networking or network function virtualization, are the enabling technologies of future internet and IT infrastructures.

These technologies are changing the characteristics and capabilities of the underlying computational substrate where services run.

As a consequence, the nature of the services that can be run on them changes too (smaller codebases, more fragmented state, service elasticity, ...).

These changes bring new requirements for service orchestrators, which need to evolve to support new scenarios where close interaction between service and infrastructure becomes essential to deliver a guaranteed level of QoS and QoE.

The extension of the Cloud to the Edge of the network through Fog Computing can significantly impact the reliability and latencies of deployed applications.

Unfortunately, the existing deployment and optimization methods pay little attention to developing and identifying complete models for

such systems, which may cause significant inaccuracies between simulated and physical run-time parameters. Existing models account for neither the inter-dependence nor the co-location of application components which causes extra communication and processing delays.

This thesis addressed these issues by carrying out experiments in both cloud and edge systems with various scales and applications in part II. In particular, two works have been presented to address (1) the placement of containers within a heterogeneous PaaS where communication-intensive applications run (chap. 4), and (2) the placement of containers to enable Cloud-Edge offloading for applications leveraging the service elasticity model (chap. 5).

Taking into account networking aspects in the decision support strategies proposed, led to exploit graph theory, and increased the complexity of the consequent allocation algorithms. In particular, the one presented in chap. 4 leveraged a quadratic integer programming model to consider inter-dependence between pods to allocate, while in chap. 5 the focus is on maximizing the utility for network operator, e.g., the inter-domain bandwidth savings.

As introduced previously, traditional Ops, both manual and automatic, consume many human resources and cannot match the rapid growth of data. AIOps emphasizes continuous learning from massive data automatically through machine learning algorithms, constantly improving and summarizing rules, and automatically making various decisions. solution is being researched. The novelty of this thesis has been to provide introductory formalizations and considerations on this new field of research that is continuously spawning from academic research to the R&D departments of the most important IT companies.

The second part of this thesis began by presenting a new reference model, *Ananke*, with data-driven parameter formulations and represen-

tations by exploiting graph theory, time-series analysis, and architecture for monitoring and tracing of communication among a cloud-native application consisting of micro-services deployed as containers.

Ananke aims to enable AIOps for future research. It is thought to be as close as possible to problems that can arise when implementing it for production-grade solutions.

Ops data is a typical nonlinear and non-stationary time-series data. Time series forecasting is the key to realize industrial intelligence. It plays a vital role in extensive applications such as information system operations, resource allocation, equipment maintenance, and AIOps.

Time series forecasting helps to grasp the future operating conditions of equipment in advance and then allocate various resources in advance or prepare counter-actions for some emergencies in advance.

Chapter 8.1 helps to understand the effect of migration of workflows and CNAs on these kinds of AIOps frameworks. The reference scenario has always been based on the K8S orchestrator, with any of the flavors currently dominating the market.

Chapter 8.1 also presented a baseline auto-scaler based on the Prophet model to forecast the amount of traffic that would lead to the event of violating SLOs, which has been continued in the following chap. 9.

It represented the synthesis of all the considerations involved in the previous works from a different perspective: deciding which action to take at run-time in order to reduce the risk of violating SLOs.

Future work will be devoted to:

- the extension of *Ananke* to better fit requirements for analysis through AI/ML tools and strategies.

- the extensions required to consider federated clouds in different data-centers.

- strategies that will take into account the dynamic environment exploited to handle multiple applications in chap 9 and manage the interference between them a two-level strategy inspired to the Mesos [40] scheduler, to account the trade-off between competition and cooperation leveraging multiple reinforcement-learning agents;
- the implementation of the proposed strategies and heuristics by the use of Kubernetes operators to explore their behavior on real workflows.

The last point also opens the discussion about the validation of researchers' work in the field of distributed and cloud computing. All the works in this thesis have been validated through (1) synthetic scenarios, (2) real traces from Alibaba Cloud [154] and Google Borg [153], and (3) real testbeds.

Validation, in these cases, is harder to accomplish, not only for the arguable complexity of the proposed strategies but also, and especially, for the lack of (1) good simulation tools and (2) the lack of real traces from production environments part of the scenarios being considered in those works.

Simulators like CloudSIM [190] are already available, but most of the time, they are not able to reproduce scenarios like the ones presented here. They also are harder to maintain and update. When writing, the last commit on the official CloudSIM repository goes back to June 2019, underlining how this simulator could lack the required models to execute a simulation of the novel environments that are arising. Instead, big companies should consider producing real traces to test against the academic environment's proposed algorithms. Companies and IT academic folks should rely more on each other. The academic journals are full of clever strategies that could optimize the efficiency of clouds. Efficiency can span from QoS and SLAs management up to cost and energy

savings. This last is strictly related to environmental issues that IT has to face up, especially now that politicians, not for pleasure, are finally interested in climate change problems.

At the same time, strategies involved in the de-facto standard orchestrators are too simple and far away to reach the goals achieved on academic papers.

The closeness between companies and academies will help to reduce this gap as well as the DevOps philosophy reduced the gap between companies' internal teams and customers.

Chapter 11

The Red Hat experience: future directions

«Talk is cheap. Show me the code.»

Linus Torvalds

One of the author's targets during the Ph.D. program has been to live an experience in a company working in the field of containers orchestration.

Despite the difficulties given by these years due to COVID-19, during the last three months of the Ph.D. program, the author pursued a remote experience at Red Hat in the Quality Engineering (QE) team in charge of OpenShift.

This experience has a twofold meaning for the author. From a scientific point of view, it allows being in touch with industrial issues related to the themes of this thesis and the work pursued during the past three years, giving the author a 360-degree view of the cloud landscape. It is especially true since Red Hat is a leader in the market of PaaS not only for enabling solutions provided by public cloud providers but also

for their private and hybrid cloud solutions: they can only be possible by leveraging solutions based on the Open Source philosophy.

The Open Source philosophy brings to the second meaning for the author, involved with the Free Software communities in Catania since high-school days: the GNU/Linux User Group, the FreakNet MediaLab, the MusIF ("Museo dell'Informatica Funzionante").

Research and science should be open and available for all people, also when people refuse it: it is trivial to think about Nicolaus Copernicus, Galileo Galilei, or, looking at today, the NoVax controversy. Fortunately, many other people today trust science, with some criticizing companies and asking not to protect vaccines by patents, making them open source essentially. One of the main reasons that brought the author to the academy lay in having the chance to give a small contribution to science, with a high interest in distributing and sharing knowledge with the world, and having some of those contributions be implemented for production-grade use, eventually.

In three only months, having the possibility to both learn and give a strong contribution from a research point of view is harder.

At the time of writing, most of the work is being done to assure main features of OpenShift (Core, SDN, Storage, Observability, ...) can be provided on ARM64 clusters.

Having support on AWS is essential to run ARM software on these clusters, but it is more important to have it running on workers at the Edge of the Network.

Edge devices deployed out in the field pose very different operational, environmental, and business challenges from those of cloud computing.

A project is currently being designed and implemented publicly: MicroShift. It is a research project that is exploring how OpenShift can be optimized for small form factor and edge computing, by allowing:

- frugal use of system resources (CPU, memory, network, storage, etc.);
- toleration of severe networking constraints;
- integration with edge-optimized operative systems, i.e., Fedora IoT and RHEL for Edge;
- self-healing to update and roll-back safely and seamlessly;
- integration with *classic* OpenShift and their APIs.

Finally, having the possibility to work with a large number of people around the world, with an extensive code base, and thousands of autonomic jobs running during each day in the continuous integration and delivery pipelines bring to another challenge in the field of the AIOps.

Continuous Integration (CI) involves running automated builds and tests of software before it is merged into a production codebase. Data produced by the large builds and tests of OpenShift are difficult to parse if someone is trying to figure out why a build is failing or why a particular set of tests is not passing.

OpenShift, Kubernetes, and a few other platforms have their CI data public and Open Source. It is real-world production operations data, a rarity for public data sets in this field, as anticipated in the previous chapter.

It represents a starting point and a first initial area of investigation for the AIOps community to tackle, together with the implementation and research about strategies discussed in this thesis.

Future work, in this field, should cultivate open-source AIOps projects by developing, integrating, and operating AI methodologies for CI, leveraging the open data available.

These data represent a rich source of information for automated triaging and root cause analysis. Unfortunately, they can be very noisy, i.e., two logs of the same type but from two different sources may be different enough that traditional comparison methods are insufficient to capture this similarity. De-noising log data, exploiting models like *Ananke*, and exploiting Machine Learning models will need further investigation and will be crucial to improve the efficiency of CI/CD pipelines for AIOps-enabled companies.

Bibliography

[1] A. Di Stefano, A. Di Stefano, G. Morana, and D. Zito, "Coope4m: A deployment framework for communication-intensive applications on mesos," in *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, June 2018, pp. 36–41.

[2] B. Steer, A. Di Stefano, R. Clegg, and F. Cuadrado, "Building distributed temporal graphs from event streams," *Proceedings of the VLDB Endowment*, vol. 11, no. 8, 2018.

[3] A. Araldo, A. Di Stefano, and A. Di Stefano, "EdgeMORE: improving resource allocation with multiple options from tenants"," in *IEEE Consumer Communications & Networking Conference (CCNC)*, Las Vegas, USA, January 2020.

[4] ——, "Resource allocation for edge computing with multiple tenant configurations," in *Proceedings of the 35th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '20, 2020.

[5] A. Di Stefano, A. Di Stefano, and G. Morana, "Scheduling communication-intensive applications on mesos," *International Journal of Grid and Utility Computing (IJGUC)*, vol. 11, no. 1, pp. 103–114, 2020. [Online]. Available: https://doi.org/10.1504/IJGUC.2020.103974

[6] ——, "Ananke: A framework for cloud-native applications smart orchestration," in *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2020, pp. 82–87.

[7] ——, "Napa: A q-learning strategy for communication optimisation and auto-scaling of micro-services on sdn-enabled clouds," *IEEE TNSM Special Issue on Smart Management of Future Softwarized Networks - Submitted*, 2021.

[8] ——, "Improving qos through network isolation in paas," *Future Generation Computer Systems - Submitted*, 2021.

[9] A. Di Stefano, A. Di Stefano, G. Morana, and D. Zito, "Prometheus and aiops for the orchestration of cloud-native applications in ananke," in *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2021.

[10] A. Di Stefano. EdgeMORE: repository. [Online]. Available: https://github.com/aleskandro/cloud-edge-offloading

[11] ——. A prometheus backfilling library. [Online]. Available: https://github.com/aleskandro/go-prometheus-backfiller

[12] ——. MORA: repository, real test-bed. [Online]. Available: https://github.com/mora-resource-allocation-edge-cloud

[13] Ananke repository. [Online]. Available: https://github.com/aleskandro/cloud-native-multiplex-monitor

[14] V. C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar, "Low level metrics to high level slas - lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments," in *2010 International Conference on High Performance Computing Simulation*, 2010, pp. 48–54.

[15] F. Schulz, "Decision support for business-related design of service level agreements," in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011, pp. 35–38.

[16] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu, "Software-defined wide area network (sd-wan): Architecture, advances and opportunities," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2019, pp. 1–9.

[17] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: Performance isolation for cloud datacenter networks," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863104

[18] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 242–253, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043164.2018465

[19] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi, "Survey of performance acceleration techniques for network function virtualization," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 746–764, 2019.

[20] H. Hawilo, M. Jammal, and A. Shami, "Network function virtualization-aware orchestrator for service function chaining placement in the cloud," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 643–655, 2019.

[21] R. Cardona, "Logical components of a vxlan bgp evpn spine-and-leaf architecture," in *The Fast-Track Guide to VXLAN BGP EVPN Fabrics*. Springer, 2021, pp. 27–65.

[22] B. A. Alrashed and W. Hussain, "Managing sla violation in the cloud using fuzzy re-schdneg decision model," in *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2020, pp. 136–141.

[23] Openslo initiative. [Online]. Available: https://openslo.com/

[24] F. Motavaselalhagh, F. S. Esfahani, and H. R. Arabnia, "Knowledge-based adaptable scheduler for saas providers in cloud computing," *Human-centric Computing and Information Sciences*, vol. 5, no. 1, pp. 1–19, 2015.

[25] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM*, 2016.

[26] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," *2017 Global Internet of Things Summit (GIoTS)*, pp. 1–6, 2017.

[27] A. Schäfer, M. Reichenbach, and D. Fey, *Continuous Integration and Automation for Devops*, 09 2013, vol. 170, pp. 345–358.

[28] P. Agrawal and N. Rawat, "Devops, a new approach to cloud development testing," in *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, vol. 1, Sep. 2019, pp. 1–4.

[29] G. B. Ghantous and A. Gill, "Devops: Concepts, practices, tools, benefits and challenges," in *PACIS*, 2017.

[30] (2019, Feb) Everything you need to know about aiops. [Online]. Available: https://www.moogsoft.com/resources/aiops/guide/everything-aiops/

[31] Y. Dang, Q. Lin, and P. Huang, "Aiops: real-world challenges and research innovations," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 4–5.

[32] Nextel: Bringing aiops to network operations. [Online]. Available: https://www.ibm.com/case-studies/nextel-networkops-video

[33] S. J. Taylor and B. Letham, "Forecasting at scale," *PeerJ Preprints*, vol. 5, p. e3190v2, Sep. 2017. [Online]. Available: https://doi.org/10.7287/peerj.preprints.3190v2

[34] Esxi. [Online]. Available: https://www.vmware.com/it/products/esxi-and-esx.html

[35] Xen hypervisor. [Online]. Available: https://xenproject.org/

[36] Cloud native persistent desktop virtualization on aws. [Online]. Available: https://aws.amazon.com/workspaces

[37] A. Busch and M. Kammerer, "Network performance influences of software-defined networks on micro-service architectures," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 153–163. [Online]. Available: https://doi.org/10.1145/3427921.3450236

[38] L. M. Vaquero, F. Cuadrado, Y. Elkhatib, J. Bernal-Bernabe, S. N. Srirama, and M. F. Zhani, "Research challenges in nextgen service orchestration," *Future Generation Computer Systems*, vol. 90, pp. 20–38, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18303157

[39] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[40] Apache mesos. [Online]. Available: http://mesos.apache.org/

[41] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 323–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972490

[42] P. U-Chupala, Y. Watashiba, K. Ichikawa, S. Date, and H. Iida, "Container rebalancing: Towards proactive linux containers placement optimization in a data center," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, July 2017, pp. 788–795.

[43] J. Monsalve, A. Landwehr, and M. Taufer, "Dynamic cpu resource allocation in containerized cloud environments," in *2015 IEEE Int. Conf. on Cluster Computing*, Sep. 2015, pp. 535–536.

[44] Kubernetes - scheduling. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

[45] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[46] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *2017 9th International Conference on Knowledge and Smart Technology (KST)*, Feb 2017, pp. 254–259.

[47] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, Dec 2017, pp. 1–8.

[48] R. DelValle, P. Kaushik, A. Jain, J. Hartog, and M. Govindaraju, "Electron: Towards efficient resource management on heterogeneous clusters with apache mesos," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, June 2017, pp. 262–269.

[49] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Operations Research*, vol. 48, no. 2, pp. 256–267, 2000. [Online]. Available: https://pubsonline.informs.org/doi/abs/10.1287/opre.48.2.256.12386

[50] Rapl. [Online]. Available: https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl

[51] T. Renner, L. Thamsen, and O. Kao, "Network-aware resource management for scalable data analytics frameworks," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 2793–2800.

[52] P. Saha, A. Beltre, and M. Govindaraju, "Exploring the fairness and resource distribution in an apache mesos environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018, pp. 434–441.

[53] D. Zhang *et al.*, "Container oriented job scheduling using linear programming model," in *ICIM*, 2017.

[54] *Bin-Packing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 426–441.

[55] Hosting rapidly scaling video applications on amazon eks clusters. [Online]. Available: https://aws.amazon.com/blogs/apn/hosting-rapidly-scaling-video-applications-on-amazon-eks-clusters/

[56] Josilo and Dan, "Wireless and computing resource allocation for selfish computation offloading in edge computing," in *IEEE INFOCOM*, 2019.

[57] A. Araldo, G. Dán, and D. Rossi, "Caching encrypted content via stochastic cache partitioning," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 548–561, Feb 2018.

[58] W. Chu *et al.*, "Joint cache resource allocation and request routing for in-network caching services," *Comp. Net.*, vol. 131, pp. 1–14, 2018.

[59] D. Zarchy *et al.*, "Capturing resource tradeoffs in fair multi-resource allocation," in *IEEE INFOCOM*, 2015.

[60] Y. Tao, X. Wang, X. Xu, and Y. Chen, "Dynamic resource allocation algorithm for container-based service computing," in *IEEE Int. Symp. on Autonomous Decentralized System (ISADS)*, March 2017.

[61] H. Mao *et al.*, "Resource management with deep reinforcement learning," in *ACM Hotnet Workshop*. ACM, 2016.

[62] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *2017 IEEE 10th Int. Conf. on Cloud Computing (CLOUD)*, June 2017, pp. 472–479.

[63] Managing resources for containers. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[64] Openshift default scheduler. [Online]. Available: https://docs.openshift.com/container-platform/3.6/admin_guide/scheduling/scheduler.html#admin-guide-scheduler

[65] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 44–48, Sep. 2012. [Online]. Available: http://doi.acm.org/10.1145/2378956.2378964

[66] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman, "Data center transport mechanisms: Congestion control theory and ieee standardization," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, Sep. 2008, pp. 1270–1277.

[67] 802.1qau – congestion notification. [Online]. Available: https://1.ieee802.org/dcb/802-1qau/

[68] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar, "Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers," in *2010 18th IEEE Symposium on High Performance Interconnects*, Aug 2010, pp. 58–65.

[69] C. Xu, K. Rajamani, and W. Felter, "Nbwguard: Realizing network qos for kubernetes," in *Proceedings of the 19th International Middleware Conference Industry*, ser. Middleware '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 32–38. [Online]. Available: https://doi.org/10.1145/3284028.3284033

[70] B. P. Rimal, M. Maier, and M. Satyanarayanan, "Experimental Testbed for Edge Computing in Fiber-Wireless Broadband Access Networks," *IEEE Commun. Mag.*, vol. 56, no. 8, pp. 160–167, 2018.

[71] M. Enguehard, G. Carofiglio, and D. Rossi, "A popularity-based approach for effective cloud offload in fog deployments," in *ITC*, 2018.

[72] S. Maheshwari *et al.*, "Scalability and performance evaluation of edge cloud systems for latency constrained applications," in *2018 IEEE/ACM Symp. on Edge Computing (SEC)*, Oct 2018, pp. 286–299.

[73] F. Bronzino *et al.*, "Lightweight , General Inference of Streaming Video Quality from Encrypted Traffic," 2019.

[74] P. Caballero, A. Banchs, G. De Veciana, and X. Costa-Perez, "Network Slicing Games: Enabling Customization in Multi-Tenant Mobile Networks," *IEEE/ACM Trans. Net.*, vol. 27, no. 2, pp. 662–675, 2019.

[75] B. Xiang, J. Elias, F. Martignon, and E. D. Nitto, "Joint Network Slicing and Mobile Edge Computing in 5G Networks," in *IEEE ICC*, 2019.

[76] S. Wang *et al.*, "Adaptive Federated Learning in Resource Constrained Edge Computing Systems," *IEEE JSAC*, vol. 37, no. 6, 2019.

[77] Y. Jin, Y. Wen, and C. Westphal, "Optimal Transcoding and Caching for Adaptive Streaming in Media Cloud," *IEEE Trans. Circ. and Sys. Video Tech.*, vol. 25, no. 12, pp. 1914–1925, 2015.

[78] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A scalable solution to the multi-resource QoS problem," in *IEEE Real Time Systems Symp.*, 1999.

[79] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014.

[80] M. T. Krieger, O. Torreno, O. Trelles, and D. Kranzlmüller, "Building an open source cloud environment with auto-scaling resources for executing bioinformatics and biomedical workflows," *Future Generation Computer Systems*, vol. 67, pp. 329 – 340, 2017.

[81] A. Carrega and M. Repetto, "Energy-aware consolidation scheme for data center cloud applications," in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 2. IEEE, 2017, pp. 24–29.

[82] N. D. Keny and A. Kak, "Adaotive containerization for microservices in distributed cloud systems," in *IEEE Consumer Communications & Networking Conference (CCNC)*, Las Vegas, USA, January 2020.

[83] P. Leitner, J. Cito, and E. Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, 2016, pp. 165–174.

[84] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 25–32.

[85] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2114–2129, 2019.

[86] H. Dai, H. Li, W. Shang, T.-H. Chen, and C.-S. Chen, "Logram: Efficient log parsing using n-gram dictionaries," 2020.

[87] C. Kan and P. Guingo, "Passive network monitoring system," Jan. 27 2009, uS Patent 7,483,379.

[88] T. Tung, F. Badruddoja, and J. C. Kang, "Cloud service monitoring system," Dec. 17 2013, uS Patent 8,612,599.

[89] P. N. Nedeltchev, A. S. Akhter, and C. M. Pignataro, "Active and passive dataplane performance monitoring of service function chaining," Apr. 7 2016, uS Patent App. 14/504,076.

[90] Splunk. [Online]. Available: https://splunk.com

[91] Sysdig. [Online]. Available: https://sysdig.com

[92] M. Catillo, M. Rak, and U. Villano, "Auto-scaling in the cloud: Current status and perspectives," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, P. Hellinckx, and J. Natwichai, Eds. Cham: Springer International Publishing, 2020, pp. 616–625.

[93] J. Ding, R. Cao, I. Saravanan, N. Morris, and C. Stewart, "Characterizing service level objectives for cloud services: Realities and myths," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*, 2019, pp. 200–206.

[94] M. Palacios, J. Garcia Fanjul, and J. Tuya, "Design and implementation of a tool to test service level agreements," *IEEE Latin America Transactions*, vol. 12, no. 2, pp. 256–261, 2014.

[95] S. K. Garg, S. Versteeg, and R. Buyya, "Smicloud: A framework for comparing and ranking cloud services," in *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, 2011, pp. 210–218.

[96] J. M. García, P. Fernández, C. Pedrinaci, M. Resinas, J. Cardoso, and A. Ruiz-Cortés, "Modeling service level agreements with linked usdl agreement," *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 52–65, 2017.

[97] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[98] D. B. Noureddine, A. Gharbi, and S. B. Ahmed, "Multi-agent deep reinforcement learning for task allocation in dynamic environment." in *ICSOFT*, 2017, pp. 17–26.

[99] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[100] T. Subramanya and R. Riggio, "Centralized and federated learning for predictive vnf autoscaling in multi-domain 5g networks and beyond," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 63–78, 2021.

[101] Checkpoint/restore in linux userspace. [Online]. Available: https://www.criu.org/

[102] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev *et al.*, "B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 74–87.

[103] X. Zuo, M. Wang, T. Xiao, and X. Wang, "Low-latency networking: Architecture, techniques, and opportunities," *IEEE Internet Computing*, vol. 22, no. 5, pp. 56–63, 2018.

[104] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, N. Dai, and H.-S. Lee, "Furion: Engineering high-quality immersive virtual reality on today's mobile devices," *IEEE Transactions on Mobile Computing*, vol. 19, no. 7, pp. 1586–1602, 2019.

[105] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 197–210.

[106] G. Cammarata, A. Di Stefano, G. Morana, and D. Zito, "Evaluating the performance of a4sdn on various network topologies," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2016, pp. 801–808.

[107] A. Di Stefano, G. Cammarata, G. Morana, and D. Zito, "A4sdn - adaptive alienated ant algorithm for software-defined networking," in *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, 2015, pp. 344–350.

[108] G. Cammarata, A. Di Stefano, G. Morana, and D. Zito, "Energy-aware routing in a4sdn," in *Conference on Complex, Intelligent, and Software Intensive Systems*. Springer, 2017, pp. 577–588.

[109] X. Liu, C. Wang, B. B. Zhou, J. Chen, T. Yang, and A. Y. Zomaya, "Priority-based consolidation of parallel workloads in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 9, pp. 1874–1883, Sep. 2013.

[110] H. Khazaei, J. Misic, and V. B. Misic, "A fine-grained performance model of cloud computing centers," *IEEE Transactions on Parallel & Distributed Systems*, vol. 24, no. 11, pp. 2138–2147, nov 2013.

[111] Redhat openshift. [Online]. Available: https://openshift.org

[112] Amazon aws. [Online]. Available: https://aws.amazon.com/

[113] Rackspace. [Online]. Available: https://www.rackspace.com/

[114] Microsoft azure. [Online]. Available: https://azure.microsoft.com/en-us/

[115] Google cloud platform. [Online]. Available: https://clod.google.com

[116] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," *Science of Computer Programming*, vol. 90, pp. 116 – 134, 2014, special Issue on Component-Based Software Engineering and Software Architecture. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313001962

[117] P. Fan, J. Wang, Z. Zheng, and M. R. Lyu, "Toward optimal deployment of communication-intensive cloud applications," in *2011 IEEE 4th International Conference on Cloud Computing*, July 2011, pp. 460–467.

[118] Kubernetes. [Online]. Available: https://kubernetes.io

[119] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *2010 IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 51–58.

[120] A. D. Stefano, G. Morana, and D. Zito, "Improving the allocation of communication-intensive applications in clouds using time-related information," in *2012 11th International Symposium on Parallel and Distributed Computing*, June 2012, pp. 71–78.

[121] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849 – 861, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17302224

[122] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The Site Reliability Workbook: Practical Ways to Implement SRE*, 1st ed. O'Reilly Media, Inc., 2018.

[123] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956–2983, 2018.

[124] A. D. Stefano, G. Morana, and D. Zito, "Ucms: User-side cloud management system," in *2013 IEEE International Conference on Communications Workshops (ICC)*, June 2013, pp. 1372–1377.

[125] U. Elsner, "Graph partitioning – a survey," 1997.

[126] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, March 1999.

[127] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Springer, 2004.

[128] D. Bienstock, "Computational study of a family of mixed-integer quadratic programming problems," in *Integer Programming and Combinatorial Optimization*, E. Balas and J. Clausen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 80–94.

[129] B. A. Julstrom, "Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 607–614. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068111

[130] P. Chu and J. Beasley, "A genetic algorithm for the multidimensional knapsack problem," *Journal of Heuristics*, vol. 4, no. 1, pp. 63–86, Jun 1998. [Online]. Available: https://doi.org/10.1023/A:1009642405419

[131] A. Hiley and B. A. Julstrom, "The quadratic multiple knapsack problem and three heuristic approaches to it," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '06. New York, NY, USA: ACM, 2006, pp. 547–552. [Online]. Available: http://doi.acm.org/10.1145/1143997.1144096

[132] E. Balas, S. Ceria, and G. Cornuéjols, "A lift-and-project cutting plane algorithm for mixed 0–1 programs," *Mathematical Programming*, vol. 58, no. 1, pp. 295–324, Jan 1993. [Online]. Available: https://doi.org/10.1007/BF01581273

[133] E. Balas and J. B. Mazzola, "Nonlinear 0–1 programming: I. linearization techniques," *Mathematical Programming*, vol. 30, no. 1, pp. 1–21, Sep 1984. [Online]. Available: https://doi.org/10.1007/BF02591796

[134] F. Glover and E. Woolsey, "Converting the 0-1 polynomial programming problem to a 0-1 linear program," *Operations Research*, vol. 22, no. 1, pp. 180–182, 1974. [Online]. Available: http://www.jstor.org/stable/169227

[135] Hey http load generator. [Online]. Available: https://github.com/rakyll/hey

[136] D-itg, distributed internet traffic generator. [Online]. Available: http://www.grid.unina.it/software/ITG/

[137] Priorities in kubernetes official source code. [Online]. Available: https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/noderesources/balanced_allocation.go

[138] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in -, 2010.

[139] S. Ghosh, R. Rajkumar, J. Hansen, and J. Lehoczky, "Scalable resource allocation for multi-processor QoS optimization," in *IEEE ICDCS*, 2003.

[140] S. Josilo and G. Dan, "Selfish Decentralized Computation Offloading for Mobile Cloud Computing in Dense Wireless Networks," *IEEE Trans. Mobile Comput.*, vol. 1233, no. c, 2018.

[141] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "Tosca: portable automated deployment and management of cloud applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[142] Openconnect service by netflix. [Online]. Available: https://openconnect.netflix.com/en/

[143] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th {USENIX} Symp. on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 689–703.

[144] Resource usage monitoring on kubernetes. [Online]. Available: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

[145] J. Liang *et al.*, "When https meets cdn: A case of authentication in delegated service," in *IEEE Symp. on Security and Privacy*, May 2014.

[146] A. Arulselvan, "A note on the set union knapsack problem," *Discrete Applied Mathematics*, vol. 169, pp. 214 – 218, 2014.

[147] Y. He, H. Xie, T.-L. Wong, and X. Wang, "A novel binary artificial bee colony algorithm for the set-union knapsack problem," *Future Generation Computer Systems*, vol. 78, pp. 77 – 86, 2018.

[148] M. M. Akbar, M. S. Rahman, M. Kaykobad, E. Manning, and G. Shoja, "Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls," *Computers & Operations Research*, vol. 33, no. 5, pp. 1259 – 1273, 2006.

[149] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*, 1st ed. Springer, 2004.

[150] D. G. Kirkpatrick and R. Seidel, "The ultimate planar convex hull algorithm?" *SIAM J. on Comp.*, vol. 15, no. 1, pp. 287–299, 1986.

[151] Y. Song, C. Zhang, and Y. Fang, "Multiple multidimensional knapsack problem and its applications in cognitive radio networks," in *MILCOM 2008 - 2008 IEEE Military Comm. Conf.*, Nov 2008, pp. 1–7.

[152] C. Pahl and B. Lee, "Containers and Clusters for Edge Cloud Architectures - a Technology Review," in *IEEE FiCloud2*, 2015.

[153] Google borg cluster usage traces. [Online]. Available: https://github.com/google/cluster-data

[154] Alibaba cluster trace program. [Online]. Available: https://github.com/alibaba/clusterdata

[155] M. Virmani, "Understanding devops bridging the gap from continuous integration to continuous delivery," in *(INTECH 2015)*, May 2015.

[156] D. N. et al., "A software architecture framework for quality-aware devops," in *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, ser. QUDOS 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 12–17.

[157] Netflix earnings, devops and profitability. [Online]. Available: https://www.onpage.com/netflix-earnings-devops-profitability/

[158] J. Humble and G. Kim, *Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations*. IT Revolution, 2018.

[159] G. B. Ghantous and A. Q. Gill, "Devops reference architecture for multi-cloud iot applications," in *2018 IEEE 20th Conference on Business Informatics (CBI)*, vol. 1. IEEE, 2018, pp. 158–167.

[160] J. Wettinger, V. Andrikopoulos, and F. Leymann, "Automated capturing and systematic usage of devops knowledge for cloud applications," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 60–65.

[161] C. Watson, "Netflix raas: Reliability as a service." Santa Clara, CA: USENIX Association, Mar. 2015.

[162] A. Jula, E. Sundararajan, and Z. Othman, "Cloud computing service composition: A systematic literature review," *Expert Systems with Applications*, vol. 41, no. 8, pp. 3809 – 3824, 2014.

[163] F. Ahmadighohandizi and K. Systä, "Icdo: Integrated cloud-based development tool for devops," in *SPLST*, 2015.

[164] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining devops automation for cloud applications using tosca as standardized metamodel," *Future Generation Computer Systems*, vol. 56, pp. 317 – 332, 2016.

[165] Jenkins. [Online]. Available: https://jenkins.io

[166] Gitlab. [Online]. Available: https://gitlab.com

[167] Docker. [Online]. Available: https://www.docker.com

[168] Ansible. [Online]. Available: https://www.ansible.com

[169] Terraform. [Online]. Available: https://terraform.io

[170] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "An exploratory study of devops extending the dimensions of devops with practices," 2016.

[171] V. Latora, V. Nicosia, and G. Russo, *"Complex Networks: Principles, Methods and Applications"*, Sept. 2017.

[172] M. Kivelä, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter, "Multilayer networks," *Journal of Complex Networks*, vol. 2, no. 3, pp. 203–271, 07 2014. [Online]. Available: https://doi.org/10.1093/comnet/cnu016

[173] B. Steer, F. Cuadrado, and R. Clegg, "Raphtory: Streaming analysis of distributed temporal graphs," *Future Generation Computer Systems*, vol. 102, pp. 453 – 464, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X19301621

[174] Cloudwatch. [Online]. Available: https://aws.amazon.com/cloudwatch/

[175] W. Conradie, V. Goranko, and C. Robinson, *Logic and Discrete Mathematics : a Concise Introduction, Solutions Manual*, 2015.

[176] The opentracing project. [Online]. Available: https://opentracing.io/

[177] promtool: Add data importers for the most requested formats. [Online]. Available: https://github.com/prometheus/prometheus/issues/7119

[178] Prometheus storage and bulk-import from openmetrics format. [Online]. Available: https://prometheus.io/docs/prometheus/latest/storage/#backfilling-from-openmetrics-format

[179] G. E. Box and Jenkins, *Time series analysis: forecasting and control*. San Francisco: Holden-Day, 1970.

[180] Flannel sdn. [Online]. Available: https://github.com/flannel-io/flannel

[181] Kuberentes - horizontal pod autoscaler. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[182] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[183] G. N. Lance and W. T. Williams, "Computer Programs for Hierarchical Polythetic Classification ("Similarity Analyses")," *The Computer Journal*, vol. 9, no. 1, pp. 60–64, 05 1966. [Online]. Available: https://doi.org/10.1093/comjnl/9.1.60

[184] S. M. Emran and N. Ye, "Robustness of chi-square and canberra distance metrics for computer intrusion detection," *Quality and Reliability Engineering International*, vol. 18, no. 1, pp. 19–28, 2002. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/qre.441

[185] S.-Y. Chen, Y. Yu, Q. Da, J. Tan, H.-K. Huang, and H.-H. Tang, "Stabilizing reinforcement learning in dynamic environment with application to online recommendation," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1187–1196. [Online]. Available: https://doi.org/10.1145/3219819.3220122

[186] M. Pieters and M. A. Wiering, "Q-learning with experience replay in a dynamic environment," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2016, pp. 1–8.

[187] Prometheus. [Online]. Available: https://prometheus.io

[188] V. Kolici, A. Herrero, F. Xhafa, and L. Barolli, "A study on the performance of oracle grid engine for computing intensive applications," in *2014 International Conference on Intelligent Networking and Collaborative Systems*, 2014, pp. 282–288.

[189] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[190] Cloudsim: A framework for modeling and simulation of cloud computing infrastructures and services. [Online]. Available: https://github.com/Cloudslab/cloudsim