Informatics

Ensuring Service Level Objective Adherence in the Edge-Cloud Continuum

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Thomas Werner Pusztai
Matrikelnummer 00727214

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Zweitbetreuung: Asst. Prof. Dr. Stefan Nastic

Diese Dissertation haben begutachtet:

Christian Becker

Vladimir Estivill-Castro

Wien, 6. Juni 2025

Thomas Werner Pusztai

TU WIEN Informatics

Ensuring Service Level Objective Adherence in the Edge-Cloud Continuum

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Thomas Werner Pusztai
Registration Number 00727214

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Second advisor: Asst. Prof. Dr. Stefan Nastic

The dissertation has been reviewed by:

Christian Becker

Vladimir Estivill-Castro

Vienna, June 6, 2025

Thomas Werner Pusztai

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Thomas Werner Pusztai

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 6. Juni 2025

Thomas Werner Pusztai

Acknowledgements

I would like to thank my advisors, Prof. Schahram Dustdar and Asst. Prof. Stefan Nastic, for giving me the opportunity to pursue a PhD and for supporting me all the way until its conclusion with discussions, feedback, and encouragement. I would also like to thank Cynthia Marcelino, Andrea Morichetta, Phillip Raith, Victor Casamyor Pujol, and all my colleagues for their friendship and support.

A very special thank you goes to my parents for always supporting me.

Finally, I would like to acknowledge the projects that have funded my PhD journey, i.e., Polaris, funded by Futurewei's Cloud Lab, RAINBOW (EU Horizon2020 grant agreement No 871403), RapidREC, funded by the Austrian Research Promotion Agency (FFG Project No. 903884), and LEOTrek, funded by NetIdee.

Kurzfassung

Das Edge-Cloud Continuum ist im letzten Jahrzehnt auf großes Interesse gestoßen. Die Möglichkeit Rechenleistung mit niedriger Latenz dort zu nutzen, wo Daten von Benutzern generiert werden und komplexe Berechnungen in die Cloud auszulagern, ermöglicht eine Vielzahl an Anwendungen, wie z.B. intelligente Fahrzeuge, Augmented Reality oder Zusammenarbeit von Menschen und Robotern in Fabriken. Um die ordnungsemäße Funktion solcher Anwendungen sicherzustellen ist es notwendig entsprechende Qualitätscharakteristika, d.h. Service Level Objectives (SLOs), zu definieren und umzusetzen.

In dieser Disseration erforschen wir die Definition und Umsetzung von SLOs im Edge-Cloud Continuum. Zuerst führen wir Abstraktionen ein, welche die Definition und Konfiguration von komplexen, Workload-spezifischen SLOs in typsicherer Weise ermöglichen. Darauf aufbauend präsentieren wir eine Middleware zur Entwicklung von Orchestrator-unabhängigen SLO Controllern, die SLOs beobachten und umsetzen. Stark typisierte Metric Querying Abstraktionen ermöglichen das Abfragen und die Aggregierung von Workload Metriken und deren Wiederverwendung als Composed Metrics. Als nächstes behandeln wir SLO-aware Scheduling von langlebigen Microservices im Edge-Cloud Continuum. Wir zeigen einen Scheduler für die SLO-aware Platzierung von asynchronen Anwendungen, die mittels eines Message Brokers kommunizieren. Danach präsentieren wir einen erweiterbaren Scheduler für die SLO-aware Platzierung von synchronen Anwendungen mit komplexen Kommunikationsabhängigkeiten, welche wir mit einem Service Graph modellieren um sie auf die aktuelle Netzwerktopologie zu mappen. Da das Edge-Cloud Continuum mehrere Cluster mit zehntausenden Nodes umfassen kann, zeigen wir einen verteilten Scheduler, der mit der Clustergröße skaliert und Schedulingkonflikte, die bei verteilten Schedulern normalerweise auftreten, niedrig hält. Zur Einhaltung von End-to-End Response Time SLOs in Serverless Workflows präsentieren wir einen Ressourcen Optimizer für Serverless Workflows, der die Größen der Function Inputs und deren Auswirkungen auf die Performance berücksichtigt. Basierend auf Function Performanceprofilen und den aktuellen Inputs wählt der Optimizer Ressourcenprofile aus, welche die Respose Time SLO des Workflows erfüllen und die Kosten minimieren. Schließlich erweitern wir das Edge-Cloud Continuum mit Low Earth Orbit Satelliten zu einem Edge-Cloud-Space 3D Continuum, das Rechenleistung für Anwendungen überall auf der Erde und für Erdbeobachtungssatelliten zur Verfügung stellen kann. Wir präsentieren eine Architektur und einen Scheduler um Serverless Workflows nahtlos im 3D Continuum auszuführen und dabei die End-to-End Response Time SLOs einzuhalten.

Abstract

The Edge-Cloud Continuum has received enormous interest from academia and industry over the last decade. The possibility to leverage low-latency computing power in proximity to where data are generated by users and to offload complex computations to Cloud datacenters has enabled a large variety of applications, such as smart vehicles, augmented reality, or human-robot collaboration in factories. To ensure proper functioning of these applications it is imperative that proper quality characteristics, i.e., Service Level Objectives (SLOs), are defined and enforced.

In this thesis we explore the definition and enforcement of SLOs in the Edge-Cloud Continuum. First, we propose a set of abstractions that enable the definition and configuration of complex workload-specific SLOs in a type-safe manner. Based on these abstractions we present a middleware for the creation of orchestrator-independent SLO controllers that monitor and enforce the aforementioned SLOs. Strongly typed metrics querying abstractions enable the retrieval and aggregation of workload metrics and the reuse of these aggregations in the form of composed metrics. Next, we discuss SLO-aware scheduling of long-lived microservices in the Edge-Cloud Continuum. We present a scheduler that is specifically designed for the SLO-aware placement of asynchronous applications that communicate through a message broker. Subsequently, we present an extensible scheduler that performs SLO-aware placement of synchronous applications with complex communication dependencies, which we model in a service graph abstraction for mapping onto the current network topology. Since the Edge-Cloud Continuum may encompass multiple clusters with tens of thousands of compute nodes, we also introduce a distributed scheduler, which is designed to scale to these cluster sizes, while keeping scheduling conflicts, which typically occur in distributed schedulers, low. To enable the adherence to end-to-end response time SLOs in serverless workflows we present a resource optimizer for serverless workflows, which is aware of function input sizes and their effects on the performance of the functions. Based on function performance profiles and the current inputs, the optimizer assigns resource profiles that meet the end-to-end response time SLO of the workflow, while minimizing costs. Finally, we extend the Edge-Cloud Continuum with low earth orbit satellites to an Edge-Cloud-Space 3D Continuum, which can deliver computational capacity to applications anywhere on Earth and to Earth Observation satellites in space. We present an architecture and a scheduler for seamlessly executing serverless workflows across this 3D Continuum while fulfilling end-to-end response time SLOs.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Over the past years Cloud computing and Edge computing have fused into the *Edge-Cloud continuum*, which connects the nodes in Cloud and Edge networks, as well as the nodes between them, into a large, seamless continuum [153]. Sensors are attached or connected to Edge devices and, typically, produce large amounts of data. End user devices, such as smartphones, smart glasses, or cars are also in many cases located at the Edge. Communication latency between a compute node and the users decreases closer to the Edge, but at the same time, computing power decreases as well, because these devices have limited capacity and are often battery-powered. Conversely, as nodes get closer to the Cloud end of the continuum, their computing capacity increases, but so does the latency to the user [207]. A distributed application can seamlessly execute its services on those nodes of the continuum that are most suitable for the particular task or even migrate services between nodes when the need arises.

The combination of having nodes with low latency to sensor data and end users and other nodes with high computing power for complex analytics enables many use cases in the Edge-Cloud continuum that would not be possible with traditional Cloud or Edge computing only. For example, smart vehicles used in autonomous driving generate lots of data every minute, data that need to be processed immediately to prevent accidents [140]. Sending all this information to the Cloud and waiting for a response may take too long to keep drivers safe and might even congest the network. Thus, the sensor data needs to be processed as close as possible to the network Edge to ensure low-latency responses. Cloud data centers receive aggregate sensor information and position data of the vehicles, so that they can predict traffic intensity, an operation that requires more computing power than Edge devices can offer. Another example are augmented reality (AR) applications [218], such as tourist guide applications. When users look through their smart glasses to get information about places in a city, the objects in their field of view need to be classified within milliseconds to retrieve adequate information about them. This must be done on Edge nodes in proximity to the users. These nodes can also store a part of the incoming images and send them to Cloud nodes that periodically retrain the Machine Learning (ML) model used for classification.

Many Edge-Cloud continuum applications adopt a *microservices architecture*, i.e., they consist of multiple services, with each service focusing on a distinct and cohesive set of tasks. Each microservice can be scaled independently from the others, i.e., additional instances can be started or stopped. A distinctive feature of microservices is that they are designed to be long-running, handle multiple requests at once, and remain running while waiting for new requests. This means that microservices consume resources even when idle, but it also allows them to respond immediately, without waiting for an instance to start up first.

After having been popular in the Cloud for some years, applications in the Edge-Cloud continuum have recently increasingly adopted a *serverless architecture* [32]. In serverless computing an application consists of one or more workflows that are made up of short-running serverless functions (less than a second up to a few minutes), with each function being responsible for a single task. Each serverless function is triggered by an event, e.g., a REST request or a file upload, and handles a single request at a time (some serverless platforms allow multiple requests per function, called intra-function parallelism, but it is debatable if this goes against serverless principles). Scaling is handled automatically by the platform, which creates new function instances to handle all incoming events. Idle function instances are stopped to free resources. While this saves costs for bursty workloads [136], it comes with the disadvantage of potentially having to start a new instance when a request arrives – this adds an additional delay to the request, called the cold-start delay [249, 141].

Regardless of the use case and architecture, an application should always meet the quality standards expected by its users. These quality standards are typically specified as Service Level Agreements (SLAs) [122], which, in turn, consist of one or more Service Level Objectives (SLOs). An SLO defines a limit or an acceptable range for an objectively measurable or computable metric, e.g., the maximum response time of a service. SLOs can be based on directly observable, simple metrics, such as response time or CPU usage, or they can be based on complex metrics that combine multiple simple metrics, such as the cost efficiency of a service [164].

How SLOs can be enforced differs depending on an application's architecture. If an SLO is violated or close to being violated, the microservices architecture allows for various actions to bring the SLO back to the desired state. We call such an action or a sequence of such actions an *elasticity strategy*. For microservices an elasticity strategy can, e.g., start or stop instances of the microservice (horizontal scaling), add or remove resources to/from existing instances (vertical scaling), and/or adjust configuration parameters. Most elasticity strategies can affect already running instances of a microservice. Complex elasticity strategies may also affect multiple microservices in an application.

In serverless computing the range of actions is more limited. Horizontal scaling is performed automatically by the underlying serverless orchestrator, as new requests arrive and since a serverless function typically executes only for a short time, changing its configuration at runtime makes no sense. Hence the configuration of serverless functions (resources and possibly application-specific settings) must be adjusted prior to creating new instances, based on monitoring information from previous executions or dedicated profiling runs.

Proper selection of the compute node that a task will execute on, referred to as *placement* or *scheduling*, plays a vital role for both, microservices and serverless functions. In the Cloud, where all nodes are relatively homogeneous and interconnected with a high speed network, the main scheduling challenges are high scheduling throughput and high node utilization. The Edge-Cloud continuum with its heterogeneous, partially battery-powered, nodes and highly diverse network connections makes scheduling much more challenging. Picking an appropriate node for a task involves ensuring that its hardware fulfills the requirements, that its network connection provides necessary speed and stability, and that it is close enough to the required data sources. Hence the choice of node greatly affects whether the application will be able to fulfill its SLOs or not.

1.2 Problem Statement

The enforcement of SLOs is central to ensuring that an application fulfills its quality requirements. Depending on the architecture of the application there are different ways to enforce the SLOs. Long-running microservices allow adaptation and scaling at runtime, but handling of complex SLOs that are based on compound metrics that are not directly observable on the system is challenging. Short-running serverless functions need to be adapted before a new instance is launched based on monitoring data from previous executions or from dedicated profiling runs. The former method may take a long time to adapt and the latter often suffers from the problem of finding a typical workload that is suitable for profiling. Both architectures benefit significantly from proper node selection during scheduling. However, the heterogeneous nodes and unstable network conditions of the Edge-Cloud continuum pose significant challenges. Additionally, the rapid growth of Edge clusters requires distributed scheduling approaches, because the clusters grow too large for being handled by a single scheduler.

1.2.1 Research Questions

We formulate the following research questions, which we will investigate in this work:

RQ1. *How can complex Cloud-native SLOs be effectively monitored and enforced for long-running workloads in containerized infrastructures at application runtime?*

Long-running microservices provide ideal conditions for monitoring SLO-related metrics and enforcing SLOs. This is because their long-lived nature allows extensive series of metrics to be collected and corrective actions to be taken at runtime if an SLO is violated.

Despite these favorable preconditions, challenges arise once SLOs go beyond simple resource usage or timing thresholds, such as average CPU usage or response time, which are used in most systems [45, 240, 191]. This is because complex SLOs are typically based on complex metrics, which are not directly observable on the system, like the cost efficiency of an application or the quality of its output [164]. Such complex metrics that are formed by aggregating or even forecasting a set of other metrics are difficult to incorporate in existing SLO frameworks, because they lack abstractions to do so or require the usage of a third-party provider for such metrics. Hence there is a need for exploring abstractions and mechanisms to obtain complex metrics and to enable their use in complex SLOs.

Once complex SLOs are available, they must be enforced. The most common scaling mechanism, or as we call it, *elasticity strategy*, is horizontal scaling [191]; vertical scaling is another easy to use option. Other options are rarely found in production systems and also in research, the two aforementioned approaches are the most common. However, the Edge-Cloud continuum offers elasticity not just in the resource dimension, but in three dimensions, i.e., resources, quality, and cost [58]. This entails that more more complex elasticity strategies that leverage the flexibility of these dimensions are possible. For example, a traffic prediction service could scale out (resource dimension) and incur more costs (cost dimension) during rush hour or it could adapt its precision settings (quality dimension) to keep costs low or it could perform a combination of horizontal scaling and precision adaptation. As for complex metrics, the lack of suitable abstractions and flexible frameworks, impedes the development of complex elasticity strategies. Thus, abstractions and frameworks are needed that cover the entire width of SLO development and elasticity.

RQ2. *How can network Quality of Service (QoS) SLOs be leveraged to improve the scheduling of long-running workloads in the Edge-Cloud continuum and how can a scheduler scale with the growing infrastructure size?*

The placement of a workload on Cloud nodes normally does not need to consider network QoS SLOs, because Cloud nodes within the same data center are interconnected with a high speed network. If multiple data centers are involved, the network already starts playing a role, because the latency to nodes in other data centers will be much higher than the latency to nodes in the same data center. When deploying an application in the Edge-Cloud continuum, the network becomes one of the most important factors for scheduling [24]. The farther nodes are located away from a backbone network, i.e., the closer they are to the Edge, the more diverse network speeds, latencies, and reliability become. While an Edge server may have a permanent high speed fiber optic Internet connection, a drone may have a fast 5G connection at some point and a slow, high latency connection at another point, depending on its current location. But some applications rely on a low latency between two or more of their components, such as augmented reality [218] or industrial Internet of Things (IoT) [247, 15]. Thus, to avoid reducing the user experience, the current network topology must be considered when scheduling the components of an application in the Edge-Cloud continuum.

Another scheduling challenge brought forth by the Edge-Cloud continuum is the handling of tens of thousands of nodes. The simplest scheduler architecture is the monolithic architecture, which is simple to design, but is limited in terms of the number of nodes and/or workloads it can handle, e.g., vanilla Kubernetes[1] is limited to a cluster size of 5,000 nodes [236]. However, scalability is an important feature of a scheduler [28]. To ensure it with an Edge-Cloud continuum with tens of thousands of nodes, a distributed scheduling approach must be used, where multiple scheduler instances operate in conjunction or completely independent of each other. Distributed architectures come with their own challenges [209], e.g, ensuring that the same resources are not assigned twice by different schedulers (scheduling conflicts) without reducing scheduling throughput too much by locking. In essence, ensuring that a scheduler system scales with the node infrastructure is a tradeoff between flexibility, speed, and avoidance of scheduling conflicts.

RQ3. *How can end-to-end response time SLOs of a workflow be used to optimize resources and placement of short-running serverless functions in the Edge-Cloud continuum?*

Serverless computing, where each request is handled by a dedicated, short-lived function instance, gives developers the freedom to concentrate on the business logic of their applications, because scaling and infrastructure management decisions are taken automatically by the platform. Ideally, serverless computing results in lower costs for bursty workloads compared to long-lived microservices [136]. However, in order to obtain this cost advantage, the resources of each function must be tuned such that they incur as little costs as possible, while fulfilling the response time SLOs. Given the large amount of available resource profiles and the complexity of real-life serverless workflows, finding the optimal resource configuration is challenging and requires appropriate optimization algorithms [134, 132].

When executing serverless workflows in the Edge-Cloud continuum on heterogeneous nodes with different network connectivity characteristics, the placement of the function instances becomes another challenging problem. A scheduler for serverless workflows for the Edge-Cloud continuum must consider the network QoS properties when selecting execution nodes for functions, such that the transfer time for input and output data does not endanger the fulfillment of the response time SLO [24, 166, 85].

1.3 Scientific Contributions

To address the aforementioned research questions we present a reliable, SLO-aware orchestrator for the Edge-Cloud continuum. An overview of its architecture is shown in Figure 1.1. The architecture is divided into multiple layers: The *Edge-Cloud Continuum* layer at the bottom represents all nodes in the computing continuum, i.e., cloud servers in datacenters, routers and access points in the network, cloudlet servers at strategic places, and all edge devices, such as smart cars, smart traffic lights, and drones. The *Deployments* layer at the top consists of long-running microservices and short-running

[1] https://kubernetes.io

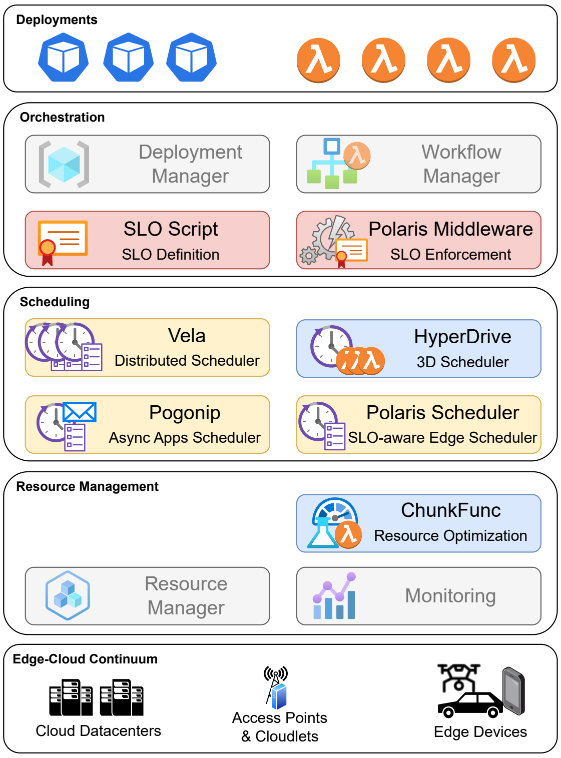


Figure 1.1: SLO-aware Orchestrator for the Edge-Cloud Continuum.

serverless functions, which are deployed in the continuum. The three layers in between make up the orchestrator. Grayed out components are not part of the contributions of this dissertation, because they can be provided by existing production-ready software. The *Orchestration* layer is responsible for managing workload deployments using the Deployment Manager, executing serverless workflows using the Workflow Manager, and enforcing SLOs. The *Scheduling* layer hosts multiple schedulers, which are specialized on specific types of workloads. The *Resource Management* layer is responsible for tracking the available and used resources on the compute nodes using the Resource Manager, optimizing resource configurations for serverless functions, and observing the performance of workloads using the Monitoring service.

In this dissertation we present on the following scientific contributions that address our research questions:

C1. Complex SLO Definition and Enforcement

With *SLO Script*, which we proposed in [184], we present high-level abstractions for the definition, monitoring, and enforcement of complex Cloud-native SLOs. SLO Script introduces the concept of a *StronglyTypedSLO*, which adds type safety to the SLO definition and configuration workflow. This ensures that an SLO can only be configured for a compatible workload and that the selected scaling mechanism, which we call elasticity strategy, is designed to work with the type of workload. A *strongly typed metrics API* boosts productivity when designing queries for metrics, which serve as the basis for SLO monitoring. Our orchestrator-independent object model enables the use of SLO script and its implementation on multiple orchestration platforms. SLO Script is discussed in Chapter 2.

To enable efficient development of controller services to enforce SLOs we introduced the *Polaris Middleware* in [183]. Its *orchestrator-independent SLO controller* concept facilitates the development of controllers that monitor and enforce SLOs defined using SLO Script. A *provider-independent SLO metrics collection and processing mechanism* extends the strongly typed metrics API from SLO Script with abstractions to define composed metrics types and instruments to provide these composed metrics in reusable manner through exchangeable controller services. We present Polaris Middleware in detail in Chapter 3.

C2. Long-running Microservices Scheduling in the Edge-Cloud Continuum

We treat scheduling in the Edge-Cloud continuum in with two complimentary subcontributions: i) SLO-aware microservice scheduling for Cloud and Edge and ii) scheduling microservices at large scale across multiple clusters.

C2.1. SLO-aware Microservice Scheduling for Cloud and Edge

When scheduling long-running microservices in the Edge-Cloud continuum, it is imperative to place them on compute nodes that meet both the resource and network QoS requirements of the application, especially if the application consists of multiple interconnected microservices. Our *Pogonip Scheduler*, which we published in [188], focuses on solving this problem for asynchronous microservice-based applications, i.e., for applications consisting of multiple microservices that communicate through a message broker. We discuss the Pogonip Scheduler in Chapter 4. *Polaris Scheduler*, introduced in [186], builds on the techniques of Pogonip and provides network SLO-aware scheduling for microservices of synchronous applications. It defines application *service graphs* that capture the dependencies between the microservices of an application and their network SLOs and the *network topology graph*, which models the network topology of the cluster with all its nodes and current network QoS properties. To perform the placement of microservices Polaris Scheduler provides an extensible, plugin-based scheduling framework, which can be adapted for additional SLOs in the future and plugins to ensure network SLO-aware scheduling in an Edge-Cloud cluster. Polaris Scheduler is presented in Chapter 5.

C2.2. Scheduling Microservices at Large Scale Across Multiple Clusters

Scheduling workloads across multiple clusters with tens of thousands of nodes in total requires a distributed scheduling approach, like the one we presented with *Vela Scheduler* in [187]. This orchestrator-independent scheduler can be distributed globally with its *3-phase scheduling workflow*, consisting of a sampling phase, a decision phase, and a commit phase. The sampling phase uses a novel two-level, informed sampling technique, called *2-Smart Sampling*, which samples from multiple globally distributed clusters. It leverages the requirements of a workload to produce samples of nodes that are very likely to be suitable for the workload. After the decision phase chooses a target node, the *MultiBind* mechanism attempts to commit the workload to that node, falling back to the second or third most suitable node if the resources on the previous node have been claimed already by another scheduler. Using this approach reduces the number of scheduling conflicts by a factor of 10. We discuss Vela Scheduler in Chapter 6.

C3. Optimizations for Serverless Functions to Fulfill End-to-End Response Time SLOs

Fulfilling the end-to-end response time SLO of a serverless workflow in the Edge-Cloud continuum requires optimizing the resource allocations of every function and their placement on the right compute nodes.

C3.1. Input- and SLO-aware Resource Optimization for Serverless Workflows

ChunkFunc, which we presented in [185], is an automatic resource configuration optimizer for serverless workflows, which, contrary to most state-of-the-art approaches, leverages information on the input size of a function to tune its resource configurations to meet the end-to-end response time SLO of the workflow while minimizing cost. ChunkFunc profiles a function with multiple typical inputs to build a performance profiles. To this end it relies on a novel *auto-tuned Bayesian Optimization approach* to reduce the number of profiling runs and infer the not explicitly profiled resource configurations using the Gaussian Process used in the Bayesian Optimization. The ChunkFunc *Workflow Optimizer* uses these performance profiles to dynamically adapt the resource profiles for the functions, based on their inputs, during the execution of the workflow. Experiments have shown an increase of SLO-adherence by a factor up to 2.78 and a cost reduction up to 61% compared to the state-of-the-art. We present ChunkFunc in Chapter 7.

C3.2. SLO-aware Scheduling of Serverless Workflows in the 3D Continuum

The HyperDrive Scheduler, introduced in [182], schedules the functions of serverless workflows on the nodes with appropriate network QoS characteristics to ensure timely transfer of data between the functions, such that the entire workflow adheres to its end-to-end response time SLO. As a first step towards future work, HyperDrive extends the Edge-Cloud continuum with low earth orbit (LEO) satellites to form a *3D continuum*. HyperDrive introduces the vision of an architecture of a *serverless platform for the 3D continuum*, which allows functions to be seamlessly executed across Cloud, Edge, and space. The *HyperDrive scheduling model* considers besides compute resources and network bandwidth and latency also the LEO-specific properties of satellite temperature and the ability to recuperate the energy needed for the function via the satellite's solar panels. The *HyperDrive heuristic scheduling algorithms* realize the scheduling model using a multi-criteria decision making approach and show a 71% lower end-to-end network latency than the best baseline scheduler in our experiments. HyperDrive is discussed in Chapter 8.

After presenting all our contributions in detail, we elaborate on related work in Chapter 9 and conclude this thesis with a summary and an outlook to future work in Chapter 10.

CHAPTER 2

SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs

This chapter introduces SLO Script, a set of abstractions and language constructs to define and configure complex workload-specific Service Level Objectives and elasticity strategies. Additionally, SLO Script provides a typed metrics API for efficient querying and aggregation of metrics and an orchestrator-independent object model to foster extensibility.

2.1 Introduction

In the previous chapter we have introduced the concept of SLOs to define the measurable bounds within which an application has to operate and elasticity strategies as a sequence of actions to be taken upon violation of these bounds to return the application to a state where the SLOs are fulfilled. The vast majority of today's cloud providers offer only support for SLOs that are based on directly observable metrics, such as CPU usage or response time. This means that customers who require a high-level SLO need to manually map it to directly measurable low-level metrics, such as CPU or memory [164].

This chapter is based on the paper T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs," in *2021 IEEE International Conference on Web Services (ICWS).*

However, from a business perspective, it is important to be able to map business goals to measurable Key Performance Indicators (KPIs), which, again, must be translated into SLOs. With a low-level average CPU usage SLO this is not easily possible. A high-level SLO that combines multiple elasticity dimensions, e.g., by combining resource usage with the total cost of the system, is better suited for this purpose.

In this chapter we continue our work envisioned in [164], which we refer to as Polaris SLO Cloud (Polaris) project, and present *SLO Script*[1], a language and accompanying framework, which permits service providers to define complex SLOs on their services and service consumers to configure and apply them to their workloads. Our main contributions with SLO Script include:

1. Novel abstractions (*StronglyTypedSLO*) with type safety features that ensure compatibility between workloads, SLOs, and elasticity strategies.
2. Language constructs: *ServiceLevelObjectives*, *ElastcityStrategies*, and *SloMappings* enable decoupling of SLOs from elasticity strategies, to promote reuse and increase the number of possible SLO/elasticity strategy combinations. Details are provided in Sections 2.3.1 and 2.3.2.
3. Strongly typed metrics API that boosts productivity when writing queries, presented in Section 2.3.3.
4. Orchestrator-independent object model that promotes extensibility, as detailed in Section 2.3.4.

The remainder of this chapter is structured as follows: Section 2.2 introduces a motivating use case to explain why SLO Script is needed and lists the research challenges and requirements for our language, Section 2.3 portrays the design and main abstractions of SLO Script, Section 2.4 describes the runtime mechanisms, Section 2.5 evaluates our SLO Script on the motivating use case on a Kubernetes implementation, and Section 2.6 summarizes our work on SLO Script.

2.2 Motivation

In the open source[2] Polaris project [164], we aim to make SLOs as the first class entities and bring multi-dimensional elasticity capabilities to the cloud computing environment. Polaris itself is part of Linux Foundation's Centaurus project[3], a novel open-source platform targeted towards building unified and highly scalable public or private distributed cloud infrastructure and edge systems.

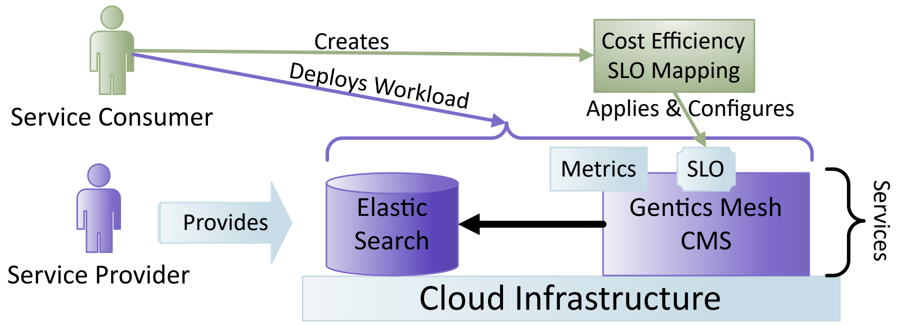


Figure 2.1: Gentics Mesh CMS cloud scenario overview.

2.2.1 Motivating Use Case

To motivate our approach, we present a real-world cloud use case, featuring a cloud service provider that wants to offer a Content Management System (CMS) in the form of Software-as-a-Service to its customers. Gentics Mesh[4] is an open source headless CMS, i.e., a CMS that is primarily used through its REST API, incorporated into a web application as a content source. The *service provider* offers customers the CMS-as-a-service for deployment on the cloud infrastructure. *Service consumers* are customers, who integrate the CMS-as-a-service into their applications. Figure 2.1 shows an overview of the use case. The deployment consists of two major components: the CMS itself and an ElasticSearch[5] database. Both need to be managed transparently for the service consumers. Each service exposes one or more metrics, e.g., CPU usage, response time, or complex metrics like cost efficiency. The service provider defines a set of SLOs that are supported by the service.

The more requests a service should be able to handle per second, the more resources it needs, and thus, the more expensive it becomes. Different service consumers have different needs with regards to requests per second and are willing to pay different prices for these guarantees. However, for most of them it is difficult, if not impossible, to specify a low-level, resource-bound SLO that delivers the best performance within their budget. This is mainly due to a lack of detailed technical understanding of the services and because a resource-bound SLO only captures a single elasticity dimension. Instead, the service consumers would prefer simply specifying a high-level *cost efficiency* of the microservices. The cost efficiency is usually defined as the number of requests per second served faster than N milliseconds divided by the total cost of the microservice [133]. To achieve this with our approach, the service consumer only needs to perform a set of simple tasks. The service consumer deploys Gentics Mesh-as-a-service – we refer to this deployment as a *workload*. To apply the cost efficiency SLO to the workload, the

[1] SLO Script is referred to as “SLO Elasticity Policy Language” in [164].
[2] https://polaris-slo-cloud.github.io
[3] https://www.centauruscloud.io
[4] https://getmesh.io
[5] https://www.elastic.co/elasticsearch/

service consumer creates an *SLO mapping*, which associates an SLO offered by the service provider with a workload of the service consumer. After creating the SLO mapping, the service consumer is finished, since the cloud will be responsible for automatically performing elasticity actions to ensure that the SLO is fulfilled.

Therefore, by allowing service consumers to specify a high-level SLO such as cost efficiency [101], our approach enables service consumers to specify a value that can be easily communicated to the non-technical, management layers of their companies, which is important for approving the budget and checking conformance with the business goals. The complex task of mapping this cost efficiency to low-level resources and performing complicated elasticity actions to achieve the SLO is left to the service provider, who knows the infrastructure and the requirements of the offered services. Using SLO Script, the service provider is able to efficiently use the know-how about the services to implement these complex SLOs.

2.2.2 Research Challenges

SLO Script addresses the following research challenges:

RC-1 *Enable complex elasticity strategies*: The majority of systems provide only simple elasticity strategies, with horizontal scaling being the most common [191]. For example, Kubernetes[6], which has the most production-level services among commonly used container orchestration systems [112], usually ships with the Horizontal Pod Autoscaler (HPA) [168]. However, some cloud providers have shown little or no further increase in application performance beyond certain instance counts [113]. Thus, a complex elasticity strategy, which, e.g., combines horizontal and vertical scaling, may achieve better results.

RC-2 *Enable high-level SLOs, based on complex metrics*: The majority of metrics used nowadays is directly measurable at the system or application level, such as CPU and memory utilization, or response time [45, 240, 191]. For example, HPA uses the average CPU utilization of all pods of a workload. We define a *composed metric* as a metric that can be obtained by aggregating and composing other metrics. In HPA they can be supplied through a custom metrics API or an external metrics API[7]. Both entail the registration of a custom API server, called an *adapter API server*, to which the Kubernetes API can proxy requests, thus, leading to additional development and maintenance effort. The custom metrics API [230] and the external metrics API [233] allow exposing arbitrary metrics (e.g., from the monitoring solution Prometheus[8]) as Kubernetes resources. However, apart from summing all values if an external metric matches multiple time series [231], the computation or aggregation of these

[6] `https://kubernetes.io`

[7] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-metrics-apis`

[8] `https://prometheus.io`

metrics must be implemented by the adapter API server. HPA allows specifying multiple metrics for scaling, but it calculates a desired replica count for each of them separately and then scales to the highest value [232]. Yet, a high-level SLO is valuable, as shown in our motivating use case.

RC-3 *Decoupling of SLOs from elasticity strategies*: If systems provide only a single elasticity strategy, e.g., horizontal scaling in HPA, it usually means that the SLO is tied to that strategy, making the system rigid and inflexible. A tight coupling between SLO evaluation and elasticity strategy in the same controller, would require re-implementing every needed SLO in every elasticity controller, leading to duplicate code and difficult maintenance. Furthermore, a specific SLO may not yet have been implemented on a certain elasticity controller, albeit being needed by a consumer.

RC-4 *Unified API for multiple metrics sources*: Each major time series database has its own query language, e.g., Prometheus has PromQL, InfluxDB[9] has Flux, and Google Cloud has MQL[10]. Thus, an implementation in a particular language ties the SLO to a certain DB, because there is no common query language for time series databases, like SQL is for relational databases.

RC-5 *Cloud vendor independence*: Common autoscaling solutions are tied to a specific orchestrator or cloud provider. All major cloud vendors have their own specific configuration of autoscaling, e.g, AWS [7], Azure [150], and Google Cloud [88] all have their own, non-portable way of configuring an autoscaler – fostering vendor lock-in. HPA, although not being tied to a particular cloud provider, is still specific to Kubernetes.

2.2.3 Language Requirements Overview

SLO Script is at the heart of the Polaris project and will ultimately support the definition and implementation of metrics, SLOs, and elasticity strategies, each of which may be generic or specifically tailored to a particular service.

An *SLO* evaluates metrics to determine whether the system conforms to the expectations defined by the service consumer. When the SLO is violated (reactive triggering) or when it is likely to be violated in the near future (proactive triggering), it may trigger an elasticity strategy. These can range from a simple horizontal scaling strategy, over more complex strategies that combine horizontal and vertical scaling, to application-specific elasticity strategies that combine scaling with adaptations of the service's configuration.

The goal is a language that presents a significant usability improvement over raw configurations that rely on YAML or JSON. To this end, the language must support higher-level abstractions than raw configuration files and provide type safety, which reduces errors and boosts productivity.

[9] https://www.influxdata.com

[10] https://cloud.google.com/monitoring/mql/reference

The requirements derived from our motivating use case and the core objectives of SLO Script are as follows:

1. Allow service consumers to configure and map an SLO to a workload.
2. Allow service consumers to choose any compatible elasticity strategy when configuring an SLO (loose coupling).
3. Allow SLOs to instantiate, configure, and trigger the elasticity strategy chosen by the service consumer.
4. Support the definition of composed metrics.
5. Support the definition of elasticity strategies.
6. Ensure compatibility between SLOs and elasticity strategies at the time of writing (i.e., type safety).
7. The SLO Script core has to be orchestrator-independent.
8. Plug into specific orchestrators using adapter libraries.
9. Service providers should be able to focus on the business logic of their metrics, SLOs, and elasticity strategies.
10. Present a DB-independent API for querying metrics.
11. Support packaging metrics, SLOs, and elasticity strategies into plugins.

SLO Script supports the use of any metrics source using adapters and elasticity strategies developed in any language, as long as their input data types match the output data types of the SLOs. This allows reusing an elasticity strategy written in a different language, e.g., because an orchestrator-specific API client may only be available in that language.

The next section will explain the design of the SLO Script language and how it achieves orchestrator-independence.

2.3 SLO Script Language Design & Main Abstractions

In this section we describe how SLO Script provides the main contributions announced in the introduction section: 1) high-level StronglyTypedSLO abstractions with type safety features, 2) constructs to enable decoupling of SLOs from elasticity strategies, 3) a strongly typed metrics API, and 4) an orchestrator-independent object model that promotes extensibility. The first two contributions are treated incrementally by the subsections 2.3.1 and 2.3.2, the third contribution is presented in subsection 2.3.3, and the fourth contribution is discussed in subsection 2.3.4.

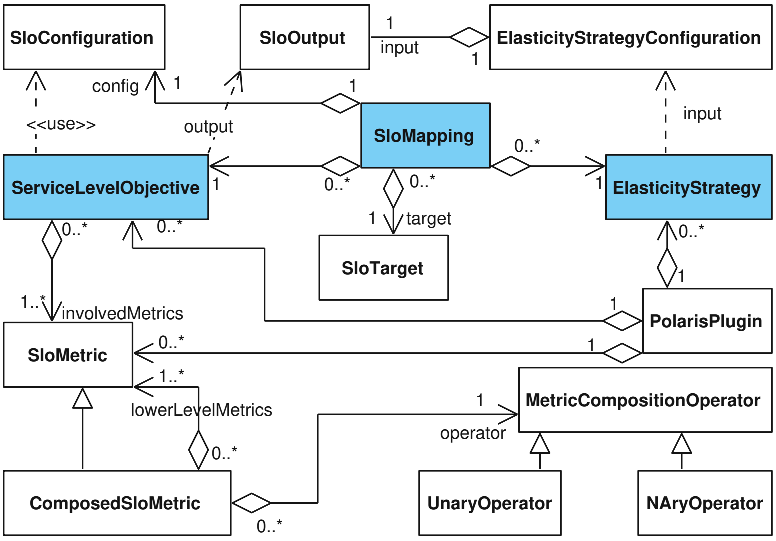


Figure 2.2: SLO Script meta-model (partial view).

2.3.1 SLO Script Overview & Language Meta-Model

SLO Script consists of high-level, domain-specific abstractions and restrictions, which constitute a language abstraction. It does not provide its own textual syntax, but uses TypeScript as its base. Using a publicly available and well-supported language, increases the chances for SLO Script to be accepted by developers and reduces maintenance effort, because language and compiler maintenance is handled by the TypeScript authors. The requirements in the previous section result in the meta-model for SLO Script, depicted as a UML class diagram, in Figure 2.2.

1) `ServiceLevelObjective` is one of the central constructs of the SLO Script language. An example instance is the `CostEfficiencySlo`, which implements the cost efficiency scenario described in Section 2.2.1. An instance of the `ServiceLevelObjective` construct defines and implements the business logic of an SLO and is configured by the service consumer using an `SloConfiguration`. The `ServiceLevelObjective` uses instances of `SloMetric` to determine the current state of the system and compare it to the parameters specified by the service consumer in the `SloConfiguration`. The metrics are obtained using our strongly typed metrics API, which abstracts a monitoring system, such as Prometheus. The metrics may be low-level metrics, directly observable on the system or higher-level metrics (instances of `ComposedSloMetric`) or a combination of both. Every evaluation of the `ServiceLevelObjective` produces an `SloOutput`, which describes how much the SLO is currently fulfilled and is used as a part of the input to an `ElasticityStrategy`. Both, `ServiceLevelObjective` and `ElasticityStrategy`, define the type of `SloOutput` they produce or require respectively, which is one of the types needed for determining compatibility among them.

2) The `ElasticityStrategy` construct represents the implementation of an elasticity

strategy. It executes a sequence of elasticity actions to ensure that a workload fulfills an SLO. Elasticity actions may include, e.g., provisioning or deprovisioning of resources, changing the types of resources used, or adapting the configuration of a service. The input to an `ElasticityStrategy` is a corresponding `ElasticityStrategyConfiguration`, consisting of the `SloOutput` produced by the `ServiceLevelObjective` and static configuration provided by the consumer.

There is no direct connection between a `ServiceLevelObjective` and an `ElasticityStrategy`, which clearly shows that these two constructs are decoupled from each other. A connection between them can only be established through additional constructs, i.e., `SloOutput` or `SloMapping`.

3) The `SloMapping` construct is used by the service consumer to establish the relationship between a `ServiceLevelObjective`, an `ElasticityStrategy`, and an `SloTarget`, i.e., the workload to which the SLO applies. The `SloMapping` contains the `SloConfiguration`, which are the SLO-specific bounds that the consumer can define, the `SloTarget`, i.e., the workload to which the SLO is applied, and any static configuration for the chosen `ElasticityStrategy`.

2.3.2 StronglyTypedSLO

When defining a `ServiceLevelObjective` using SLO Script's StronglyTypedSLO mechanism, the service provider must first create an `SloConfiguration` data type that will be used by the service consumer to configure the `ServiceLevelObjective` and an `SloOutput` data type to describe its output. While each `ServiceLevelObjective` will likely have its own `SloConfiguration` type, it is recommended to reuse an `SloOutput` data type for multiple `ServiceLevelObjectives` to allow for loose coupling between `ServiceLevelObjectives` and `ElasticityStrategies`.

To create the actual SLO, a service provider must instantiate the `ServiceLevelObjective` meta-model construct, represented by the `ServiceLevelObjective` TypeScript interface. It takes three generic parameters to enable type safety: `C` denotes the type of `SloConfiguration` object that will carry the parameters from an `SloMapping`, `O` is the type of `SloOutput`, which will be fed to the elasticity strategy, and `T` is used to define the type of target workload the SLO supports. An `ElasticityStrategy` uses the same mechanism to define the type of `SloOutput` that it expects as input.

Figure 2.3 illustrates how the type safety feature of SLO Script works. There are two sets of types: those determined by the `ServiceLevelObjective` and those determined by the `ElasticityStrategy`. The `ServiceLevelObjective` defines that it needs a certain type of `SloConfiguration` (indicated by the yellow color) as configuration input. The `SloConfiguration` defines the type of `SloTarget` (orange), which may be used to scope the SLO to specific types of workloads. The `ElasticityStrategy` defines its type of `ElasticityStrategyConfiguration` (purple), which, in turn, specifies the type of `SloOutput` (blue) that is required by the `ElasticityStrategy`.

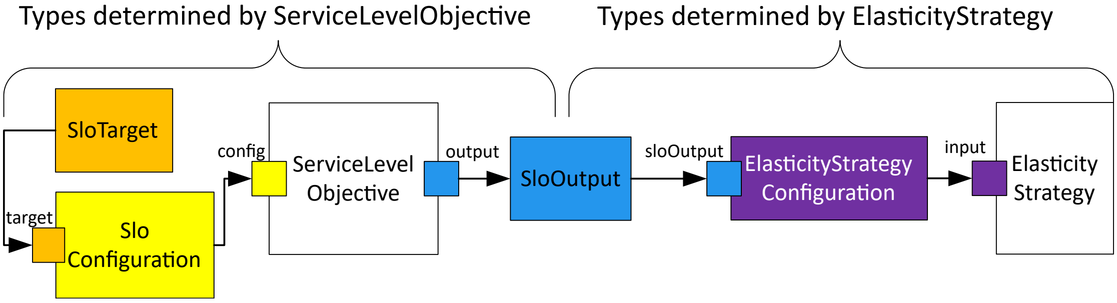

Types determined by ServiceLevelObjective
Types determined by ElasticityStrategy
SloTarget
target
Slo Configuration
config
ServiceLevel Objective
output
SloOutput
sloOutput
ElasticityStrategy Configuration
input
Elasticity Strategy

Figure 2.3: Type safety provided by StronglyTypedSLO.

Thus, the bridge between these two sets is the `SloOutput` type. Once the service consumer has chosen a particular `ServiceLevelObjective` type, the possible `SloTarget` types are fixed because of the `SloConfiguration`. Since the `ServiceLevelObjective` defines an `SloOutput` type, the set of compatible elasticity strategies is composed of exactly those `ElasticityStrategies` that have defined an `ElasticityStrategyConfiguration` with the same `SloOutput` type as input.

Type checking is especially useful in enterprise scenarios, where hundreds of SLOs need to be managed. Using YAML or JSON files for this purpose provides no way of verifying that the used SLOs, workloads, and elasticity strategies are compatible, while SLO Script provides this feature. Furthermore, using a type safe language yields significant time savings when a set of SLOs and their mappings need to be refactored.

The SLO Script runtime invokes the SLO instance at configurable intervals to check if the SLO is currently fulfilled or if the elasticity strategy needs to take corrective actions. It may simply check if the metrics currently match the requirements of the SLO or it can use predictions and machine learning to determine if the SLO is likely to be violated in the near future and thus take proactive actions through the elasticity strategy. The result of this operation is an instance of the defined `SloOutput` type, which is returned asynchronously.

2.3.3 Strongly Typed Metrics API

The strongly typed metrics API provides two types of abstractions: i) *raw metrics queries* for querying time series databases independent of the query language they use natively and ii) *composed metrics* for creating higher-level metrics from aggregated and composed lower-level metrics obtained through raw metrics queries. Since our API is based on objects, rather than on a textual language, it also comes with type safety features. When using PromQL or Flux directly, developers often need to write queries as plain strings in their application code, thereby breaking the type safety of that code. Figure 2.4 shows a class diagram with a simplified view of our strongly typed metrics API, whose raw metrics query abstractions were inspired by PromQL, with some influences from Flux, and MQL.

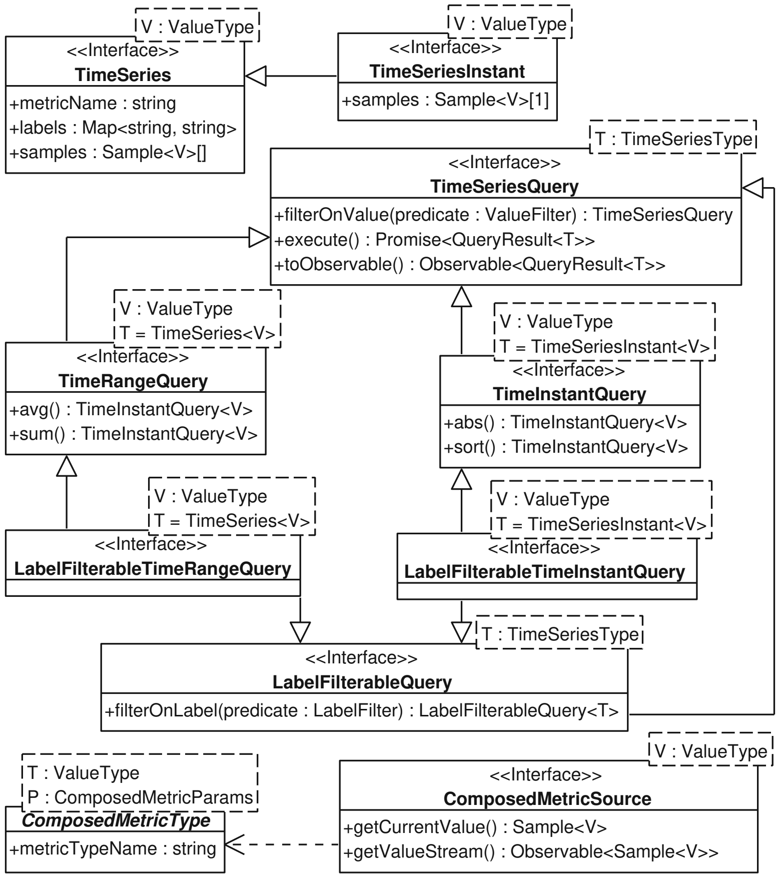


Figure 2.4: Strongly Typed Metrics API (simplified view).

For raw metrics queries, the central model type is `TimeSeries`, which describes a sequence of sampled values for a metric. In addition to the `metricName`, a `TimeSeries` has a map of `labels` that can be used to further describe its samples, e.g., a metric named `http_requests_per_sec` could have a label `service`, which identifies the particular service from which this metric was observed.

The base interface for querying time series is `TimeSeriesQuery`. Like a relational DB query results in a set of one or more rows, a time series DB query results in a set of one or more time series, each with a distinct metric name and labels combination. A query for `http_requests_per_sec` could result in two distinct `TimeSeries`, one with the label `service = 'gentics_mesh'` and one with the label `service = 'elasticsearch'`. This is why the execution of a `TimeSeriesQuery` results in a `QueryResult`, which can contain multiple `TimeSeries` instances.

A time series DB not only allows retrieving a time series with particular properties, but also allows applying functions to the data, such as various types of aggregations or sorting. Certain functions, such as aggregations, require time series with multiple samples as input, while other functions, e.g., sorting, only work on time series with a single sample. For example, one may first query all time series for `http_requests_per_sec`, then compute the sum for each single time series, and finally sort the results to see which service gets the most requests. Prometheus will return an error when trying to sort time series with multiple samples, but it requires a developer to try to execute the query first.

The `TimeSeriesInstant` model type represents time series that are limited to a single sample. To support both time series types, the `TimeSeriesQuery` interface is extended by multiple subinterfaces: `TimeRangeQuery` for queries that result in a set of `TimeSeries` and `TimeInstantQuery` for queries that result in a set of `TimeSeriesInstants`. Each of these interfaces exposes only methods for DB functions that are applicable to the respective time series type. A function may also change the time series type, e.g., `sum()` is applied to a `TimeSeries`, but it returns a `TimeSeriesInstant`.

`LabelFilterableQuery` is another subinterface of `TimeSeriesQuery` that allows applying filters on the labels. Since our metrics query API needs to produce valid DB-specific queries, label filtering is a capability of a query that is lost after applying the first DB function, e.g., `sum()`, due to the structure of PromQL queries.

A composed metric is designated by a `ComposedMetricType`. It defines the name of the composed metric, the data type used for its values, and which parameters are needed to obtain it (e.g., the name of the target workload). The metric values are supplied by a `ComposedMetricSource`, which may use raw metrics queries internally to obtain and aggregate multiple lower-level metrics, which are composed to form the higher-level composed metric.

For each `ComposedMetricType` there may be multiple `ComposedMetricSources`. This allows decoupling the type of a composed metric from the implementation that computes it and enables multiple implementations, which can be tailored to various types of workloads, such as REST APIs or databases, while delivering the same type of composed metric.

2.3.4 SLO Script Object Model

The SLO Script object model, a subset of which is shown in Figure 2.5, is an instantiation of the language's meta-model in the framework. This abstract object model allows SLO Script to achieve orchestrator independence and promotes extensibility.

Every object that is submitted to the orchestrator must be of type `ApiObject` or a subclass of it. It contains an `objectKind` attribute that describes its type. The `ObjectKind.group` attribute denotes the API group of the type, which can be seen as a package in UML. The `version` attribute identifies the version of the API group and `kind` conveys the name of the type. `ApiObject` also has a `metadata` attribute, which contains additional information about the object, including the name of the instance. The `spec` attribute contains the actual "payload" content of the object. `ObjectReference` extends `ObjectKind` with a `name` attribute to be able to reference existing object instances in the

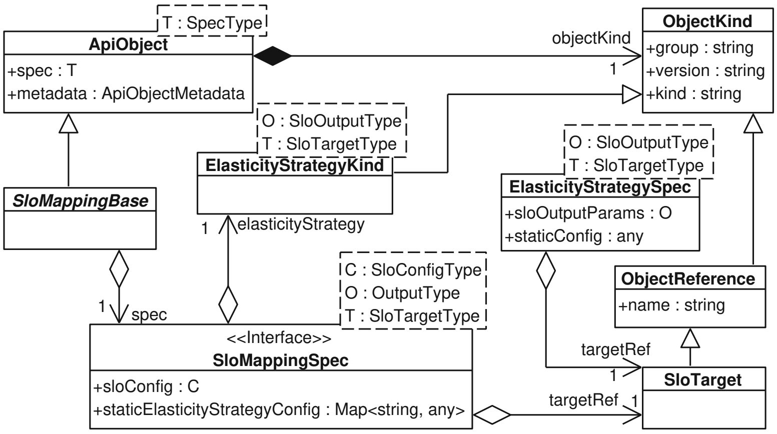


Figure 2.5: Core SLO Script Object Model Types (partial view).

orchestrator This is needed to refer to the target workload of an SLO in an `SloTarget`, which derives from `ObjectReference`.

`ApiObject` is the root extension point for objects that need to be stored in the orchestrator. For example, to instantiate the `SloMapping` construct from the meta-model, a TypeScript class needs to be created that inherits from `SloMappingBase`, which derives from `ApiObject`. It contains the type information for the spec and sets up the correct `ObjectKind` for this `SloMapping`. The `SloConfiguration` construct can be represented by an arbitrary TypeScript interface or class, but needs to be wrapped in a class implementing `SloMappingSpec`, which will store the configuration. A concrete example will be shown in Section 2.5. The `SloMapping` represents a custom resource type that needs to be registered with the orchestrator. As part of future work, we will create a build system capable of automatically generating definitions for these resources.

To identify an `ElasticityStrategy` when configuring an `SloMapping`, the `ObjectKind` subclass `ElasticityStrategyKind` is used. For each `ElasticityStrategy`, an `ElasticityStrategyKind` subclass has to be created and parameterized with the `SloOutput` and `SloTarget` types expected by the `ElasticityStrategy`. This in conjunction with the `SloOutput` type configured on a `ServiceLevelObjective` and its corresponding `SloMapping`, enables the type checking discussed in the previous sections.

The `SloOutput` meta-model construct is instantiated by creating an arbitrary TypeScript class. To allow compatibility between as many SLOs and elasticity strategies as possible, generic `SloOutput` data types, which are supported by multiple `ServiceLevelObjectives` and `ElasticityStrategies` are recommended. The `SloCompliance` class provided by the core object model conforms to this requirement. It expresses the current state of the SLO as a percentage of conformance. A value of 100% indicates that the SLO is precisely met. A higher value indicates that the SLO is violated and that additional resources, e.g., scaling out, are needed, whereas a value below 100% indicates that the SLO is being outperformed, i.e., a reduction of resources, e.g., scaling in, should be considered.

Every orchestrator has its own set of abstractions – thus, an independent framework must provide a mechanism for transforming objects between its own structure and the native structure of each supported orchestrator. To this end, SLO Script provides a transformation service that allows each orchestrator-specific connector library to register transformers for those object types that require transformation, while directly copying those objects that do not require any transformation. The transformation is not limited to the type of the root object, instead the appropriate transformer is applied to every nested object recursively.

The transformation service does not serialize to the data format required by the orchestrator (e.g., JSON or YAML). It transforms the instances of the orchestrator-independent SLO Script classes into plain JavaScript objects, which can be serialized by the orchestrator-specific connector library. The deserialization of objects received from the orchestrator is also left to the connector library. It needs to supply plain JavaScript objects to the transformation service, which transforms the objects and creates instances of SLO Script classes.

The SLO Script object model is heavily influenced by that of Kubernetes, but the two are not equal. For example, in Kubernetes there is no `objectKind` property on an object returned from the API. Instead, a Kubernetes API object contains an `apiVersion` and a `kind` property, with the former being a combination of the SLO Script `group` and `version` attributes of `ObjectKind` and the latter being equal to `ObjectKind.kind`.

To transform an SLO Script `ObjectKind` into its corresponding Kubernetes version, the SLO Script Kubernetes connector library registers a transformer for `ObjectKind`, which returns a plain JavaScript object with the `group` and `version` attributes combined into a single `apiVersion` attribute and a copy of the `kind` attribute. This alone is not enough because Kubernetes objects do not contain an `ObjectKind` property. Thus, the Polaris Kubernetes connector library also registers a transformer for the `ApiObject` class, which uses the transformation service to first transform the `ObjectKind` object and then embeds the contents of the result into a new object, which is going to become the final transformed `ApiObject`. The objects stored in the `metadata` and `spec` attributes are also first transformed and then stored in the result object. The transformation process will be explained in detail in Section 3.4.1.

The next Section will explore the runtime facilities, which are responsible for executing the defined SLOs.

2.4 Runtime Mechanisms

Technically, the cluster component used for handling an SLO is a controller for the custom resource type defined by the `SloMapping` of that SLO. The controller watches the custom resource type instances in its deployment scope, creates and destroys SLO

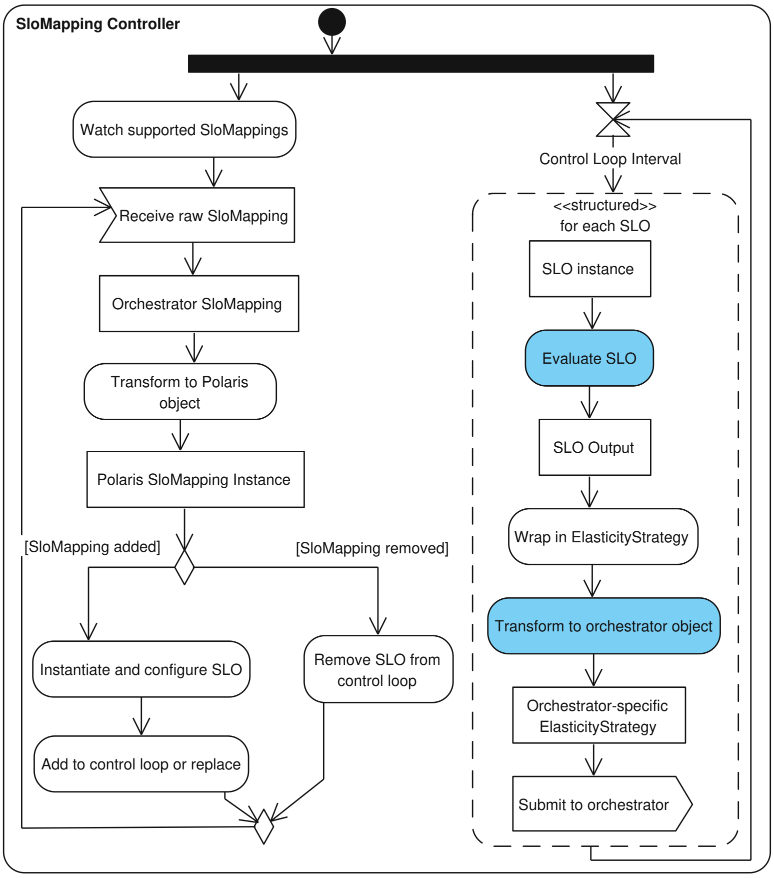

SloMapping Controller
Watch supported SloMappings
Receive raw SloMapping
Orchestrator SloMapping
Transform to Polaris object
Polaris SloMapping Instance
[SloMapping added]
[SloMapping removed]
Instantiate and configure SLO
Add to control loop or replace
Remove SLO from control loop
Control Loop Interval
<<structured>>
for each SLO
SLO instance
Evaluate SLO
SLO Output
Wrap in ElasticityStrategy
Transform to orchestrator object
Orchestrator-specific ElasticityStrategy
Submit to orchestrator

Figure 2.6: `SloMappingController` workflow and SLOs lifecycle.

class instances accordingly, and evaluates them at a defined interval. Figure 2.6 shows a UML activity diagram with the workflows within the controller.

To handle SLOs, the SLO Script runtime provides a control loop interface and a default implementation that maintains the set of active SLOs and evaluates them at a configurable interval. To add an SLO to the control loop, an `SloMapping`, which is received from the orchestrator, is needed, along with a key to uniquely identify that SLO. The key can be generated from the `metadata` of the `SloMapping` object. The `SloMapping` is used by the control loop to identify which SLO class to instantiate and to subsequently configure that instance, before adding it to its internal set.

The runtime aims to handle as many managerial tasks as possible to allow service providers to focus on their business logic. In the control loop in Figure 2.6 only the actions highlighted in blue need to be implemented by the service provider.

The control loop is designed to work on all orchestrators. It needs to be configured with an `SloEvaluator`, which handles the execution of the SLO and the subsequent submission of its output to the orchestrator. Its `evaluateSlo(key, slo)` method, gets the SLO object's key and the object itself as parameters and has to asynchronously notify its caller when the SLO evaluation is finished and the results have been submitted to the orchestrator. The runtime provides an abstract class to handle the evaluation of the SLO, as well as the wrapping of its result into the configured `ElasticityStrategy` object. It provides hooks for the orchestrator specific connector to execute code before and after the evaluation to apply the SLO's results to the orchestrator. The default implementation of the control loop gracefully handles errors during SLO evaluation, to ensure that a faulty implementation of one SLO does not prevent other SLOs in the same controller from being evaluated. Apart from an `SloEvaluator`, the SLO control loop must be configured with an `Observable` to define the evaluation interval – it must emit whenever the control loop should execute an iteration. This may be used to not only trigger a loop iteration at regular intervals, but can include other triggers as well, e.g., a "force evaluation now" event.

To be able to operate, the SLO control loop has to be integrated into a controller for the respective `SloMapping(s)`. This controller part depends heavily on the target orchestrator and should be implemented in the corresponding SLO Script orchestrator connector library. We currently provide a connector library for Kubernetes, which relies on kubernetes-client[11], the officially supported JavaScript client library for Kubernetes. Our controller implementation uses the `watch`[12] functionality of the Kubernetes API to be efficiently notified whenever a resource of an observed type is added, removed, or changed, such that the SLO control loop can be adjusted. For the strongly typed metrics API we currently provide a connector for PromQL.

The controller uses the transformation service to convert between orchestrator-independent and orchestrator specific objects. To this end, the transformation service wraps the open-source library class-transformer[13], which provides most of the facilities needed for transformation. The registration of transformers for specific classes uses custom SLO Script mechanisms. Unlike class-transformer, it allows registering a transformer not just for a single property, but globally for all instances of a class and optionally its subclasses, and use that transformer on all transformable properties of that type.

Similar to class-transformer, SLO Script utilizes a TypeScript decorator to designate the type of a class property for transformation. This is necessary because the information about the types of class properties is not available at runtime, so it needs to be attached to the constructor function object as custom ECMAScript metadata. The need for metadata that is available at runtime is also the reason why most SLO Script framework types are classes instead of TypeScript interfaces – interfaces do not exist at runtime and thus, cannot be used for carrying metadata. SLO Script's `@PolarisType` decorator sets this type metadata for a property of a class and registers a helper with class-transformer,

[11] https://github.com/kubernetes-client/javascript

[12] https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes

[13] https://github.com/typestack/class-transformer

responsible for looking up the registered SLO Script transformer for that type or using the default transformer.

Registering a transformer is also possible for specific `ObjectKind` configurations. This is used, e.g., to automatically instantiate the correct class when an object of kind `slo.polaris-slo-cloud.github.io/v1/CostEfficiencySloMapping` needs to be transformed.

To facilitate adoption of SLO Script, our project includes a Command-Line Interface (CLI) tool that can be used to generate skeletons for `SloMapping` classes and SLO controllers, as well as build and deploy them. More information about the CLI can be found in a demo video online[14].

Further details on the runtime will be presented in the next chapter. The next Section will examine the realization of our motivating use case to evaluate SLO Script.

2.5 Evaluation & Implementation

We have implemented the core library of SLO Script, its CLI, as well as the controllers and connectors for Kubernetes and Prometheus using TypeScript and Go. The motivating use case is realized using the CLI and these libraries. More details and the source code can be found in our code repositories[15].

To evaluate SLO Script we use an approach based on the guidelines defined in [155]. We use the real-world cost efficiency use case from Section 2.2.1 to illustrate that SLO Script fulfills its requirements by improving code *reusability*, *flexibility*, *ease of use*, and *expressiveness* and reducing *susceptibility to errors* and thus, increases productivity.

The goal of the cost efficiency SLO is to trigger an elasticity strategy whenever the current cost efficiency deviates too far from the defined target value. Since SLO Script allows the use of an arbitrary elasticity strategy, as long its input parameter type matches the SLO's output, we will use the term *increase resources* to refer to any sequence of elasticity actions that enlarge the resources allocated to a service, e.g., scaling up or scaling out, and *decrease resources* to any sequence of elasticity actions that reduce the resources allocated to a service, such as scaling down or scaling in.

Contrary to a simple CPU utilization SLO, it is not possible to derive whether an increase or a decrease in resources is needed by examining only the current and target values of the cost efficiency. For example, a low cost efficiency is ambiguous – it may indicate that either

- the system cannot handle the current high demand in time and that an increase in resources is needed or

[14] https://www.youtube.com/watch?v=3_z2koGTExw

[15] https://polaris-slo-cloud.github.io

- all requests are handled in time, but too many resources are provisioned compared to the few incoming requests, such that a decrease in resources is needed.

An *expressive* language is needed to distinguish these two cases. In SLO Script, we define the type `CostEfficiencySloConfig` as shown in Listing 2.1. To handle the ambiguity problem we just described, we add an additional parameter to this configuration type: the minimum percentile of requests that should be handled within the time threshold. If the number of requests per second faster than the threshold is below that percentile, the service does not have enough resources to handle the load, whereas if it is above that percentile, the service has too many resources.

```
export interface CostEfficiencySloConfig {
  responseTimeThresholdMs: number;
  targetCostEfficiency: number;
  minRequestsPercentile?: number; }
```

Listing 2.1: Cost efficiency SLO configuration.

Listing 2.2 shows the `SloMappingSpec` and `SloMapping` classes. The spec class defines in the generic parameters for its superclass that the configuration type for this SLO will be `CostEfficiencySloConfig`, the output type will be `SloCompliance`, and the target workload must be of type `RestServiceTarget`. This short definition ensures that i) the SLO can only be applied to workloads of the correct type, i.e., workloads that expose the required metrics, and that ii) only an elasticity strategy that supports the SLO's output data can be used, because each `ElasticityStrategyKind` needs to specify the compatible input types in an analogous way. This greatly reduces the possibility for deploy-time or runtime errors, because SLO Script enforces that only compatible workloads and elasticity strategies are used.

```
export class CostEfficiencySloMappingSpec extends
    SloMappingSpecBase<CostEfficiencySloConfig, SloCompliance,
    RestServiceTarget> { }
export class CostEfficiencySloMapping extends
    SloMappingBase<CostEfficiencySloMappingSpec> {
  constructor(initData?:
      SloMappingInitData<CostEfficiencySloMapping>){
    super(initData);
    this.objectKind = new ObjectKind({
      group: 'slo.polaris-slo-cloud.github.io',
      version: 'v1',
      kind: 'CostEfficiencySloMapping' });
    initSelf(this, initData);
  }
  @PolarisType(() => CostEfficiencySloMappingSpec)
  spec: CostEfficiencySloMappingSpec; }
```

Listing 2.2: Cost efficiency SLO mapping.

The constructor of the `CostEfficiencySloMapping` class initializes the `objectKind` property to ensure that the correct API group and kind are configured and uses the `@PolarisType` decorator to set the appropriate class for the transformation of the `spec` property. At the moment, the Kubernetes Custom Resource Definition (CRD) for registering the SLO mapping type with the orchestrator must either be written manually or be generated from an equivalent data structure written in Go – this will be addressed in a future version of the CLI, which will support automatic generation of CRDs.

The `CostEfficiencySlo` class implements the actual SLO. It uses the `MetricsSource` to retrieve the metrics for the target workload and uses them in conjunction with the configuration to compute an `SloCompliance` that indicates if the resources need to be increased or reduced.

To apply the SLO to a workload, service consumers need to instantiate the `SloMapping` as shown in Listing 2.3. Any TypeScript compatible IDE can provide code completion for the required properties, which greatly benefits the *ease of use*, and give immediate feedback if the chosen target workload or elasticity strategy are not compatible with the SLO, thus, *revealing errors at the time of writing*, which would have been discovered only at deploy-time or even at runtime, if plain JSON or YAML had been used for configuration.

```
export default new CostEfficiencySloMapping({
  metadata: new ApiObjectMetadata({ name:
     'data-service-cost-efficiency' }),
  spec: new CostEfficiencySloMappingSpec({
    targetRef: new RestServiceTarget({
      group: 'apps',
      version: 'v1',
      kind: 'Deployment',
      name: 'data-service' }),
    elasticityStrategy:
      new HorizontalElasticityStrategyKind(),
    sloConfig: {
      responseTimeThresholdMs: 400,
      targetCostEfficiency: 1000,
      minRequestsPercentile: 90 } }) });
```

Listing 2.3: Applying the cost efficiency SLO to a workload.

Since TypeScript is a superset of JavaScript, a developer can circumvent the type checking of SLO Script by writing plain JavaScript. The type safety can also be evaded by applying plain JSON or YAML configuration to the orchestrator. However, this is not an issue, because our aim is not to lock someone into a type safety system that cannot be circumvented in any way. The goal is to provide a language, consisting of domain-specific abstractions and restrictions, which, if used, increase productivity and provide type safety.

From the architectural perspective, the biggest benefit of SLO Script is the decoupling

Table 2.1: Lines of Code (excl. comments and blanks).

Component	Lines of Code	Generated	% of Total
SLO Mapping Type	53	50	2%
SLO Controller	224	99	8%
Runtime	2616	–	90%
Total	2893	149	100%

of SLOs from elasticity strategies, which increases code *reusability* and *flexibility*. The clear separation of SLO implementations from elasticity strategy implementations allows them to be reused in multiple combinations as long as their output/input types match. The input/output types can be seen like interfaces in object-oriented programming. If an elasticity strategy implements the interface required by the SLO, the two may be used in conjunction. This brings the flexibility of object-oriented programming to the management of SLOs in the cloud.

The SLO Script runtime eases the development of SLOs, because it lets service providers focus on their data types and business logic. The runtime's SLO control loop handles the integration with the orchestrator, as well as the management of the active SLOs. Out of the steps in the control loop, depicted in Figure 2.6, only the "Evaluate SLO" step needed to be implemented for our cost efficiency use case. This is evident from Table 2.1, which shows the line counts of the various components of our cost efficiency implementation. The SLO Script runtime makes up 90% of the total code. The `CostEfficiencySloMapping` class and its supporting types add up to 53 lines, however, 50 of these were generated by the CLI. The SLO controller and all its metrics queries take up 224 lines, 99 of which were generated.

The orchestrator independent object model of SLO Script eases the porting of SLOs and their mappings to other orchestration platforms, promoting flexibility, limiting the possibility for vendor lock-in for consumers, and fostering open source collaboration on SLOs for multiple platforms. Many SLOs may be implemented in a completely orchestrator-independent manner as well, allowing the creation of "standard SLO libraries" for instant reuse on other platforms.

2.6 Summary

This chapter has presented SLO Script, a language and accompanying framework for defining and implementing Service Level Objectives, based on TypeScript and being part of the open source Polaris project. We have motivated why SLO Script is needed using a real-world use case of a headless CMS that is used through its REST API and that should scale based on a high-level cost efficiency SLO. The major requirements for the design of SLO Script are allowing the definition and configuration of complex SLOs and elasticity strategies, as well as the definition of composed high-level metrics, decoupling of SLOs and elasticity strategies, type safety of all abstractions, and orchestrator independence.

We showed the language's meta-model and then described SLO Script's design and how it fulfills our main contributions of

1. high-level StronglyTypedSLO abstractions with type safety features,
2. decoupling of SLOs from elasticity strategies,
3. a strongly typed metrics API, and
4. an orchestrator-independent object model that promotes extensibility.

Next, we explained how SLO Script's runtime mechanisms and the SLO control loop work. For evaluating our language and framework, we illustrated how to implement and configure the cost efficiency SLO for the motivating use case and highlighted the benefits of using SLO Script and its CLI for this purpose.

CHAPTER 3

A Novel Middleware for Efficiently Implementing Complex Cloud-Native SLOs

Polaris Middleware builds on the abstractions introduced in the previous chapter by SLO Script and adds mechanisms for implementing orchestrator-independent SLO controllers, which are decoupled from the elasticity strategies they can trigger. The middleware's provider-independent SLO metrics collection and processing mechanism enables efficient querying of time-series metrics and aggregating them into reusable higher-level metrics, while the Polaris CLI tool allows for fast bootstrapping and creation of Polaris-based projects.

3.1 Introduction

In the previous chapter we introduced SLO Script's type-safe and orchestrator-independent abstractions for SLOs, metrics, and elasticity strategies. To efficiently adjust the elasticity of a deployed cloud application, which we refer to as a *workload*, based on its SLOs, a Monitor Analyze Plan Execute (MAPE) loop [142] can be implemented: i) the *monitoring* of system and workload metrics can be handled by tools, such as Prometheus[1], ii) the *analysis* of the metrics to evaluate whether the defined goals are met, is the task of an SLO, iii) the *planning* of actions to correct a violated SLO needs to be done by the

This chapter is based on the paper T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "A Novel Middleware for Efficiently Implementing Complex Cloud-Native SLOs," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[1] `https://prometheus.io`

elasticity strategy, and iv) the *execution* of the planned actions is carried out by the cloud orchestrator, e.g., Kubernetes[2].

In this chapter, we focus on the realization of SLOs, i.e., the *analysis* step of the control loop. In the analysis step, an SLO must obtain one or more metrics from the monitoring step and pass its evaluation result on to the planning step, i.e., an elasticity strategy – this is not trivial. A common approach is to implement the SLO as a control loop itself. The variety of monitoring solutions and databases (DBs) makes obtaining metrics difficult without tying the implementation to a particular vendor. Once the metrics have been obtained, they may need to be aggregated to gain deeper insights. When the current status of the SLO has been determined, the outcome needs to be conveyed to an elasticity strategy; ideally multiple elasticity strategies should be supported.

To facilitate the implementation of complex SLOs, we present the Polaris Middleware. Its implementation is published as open source, as part of the Polaris project[3]. Our main contributions with the Polaris middleware include:

1. An *orchestrator-independent SLO controller* periodically evaluates SLOs and triggers elasticity strategies, while ensuring that SLOs and elasticity strategies remain decoupled to increase the number of possible SLO/elasticity strategy combinations.

2. A *provider-independent SLO metrics collection and processing mechanism* allows querying raw time series metrics, as well as, composing multiple metrics into reusable higher-level metrics.

3. A *CLI Tool* creates and manages projects that rely on the Polaris middleware.

Additionally, we provide platform connectors for Kubernetes, which has been found to have the most capabilities for production-level services among commonly used container orchestrators [112], and Prometheus, which is a popular choice for a time series DB.

The remainder of this chapter is structured as follows: Section 3.2 provides further motivation for our work using a real-world use case, Section 3.3 presents a high-level overview of the Polaris middleware, Section 3.4 describes the central mechanisms, and Section 3.5 their implementation. In Section 3.6 we evaluate the Polaris middleware by implementing the real-world use case and in Section 3.7 we summarize this chapter.

3.2 Motivation

To motivate the need for the Polaris middleware, we revisit the illustrative scenario from the previous chapter.

[2] https://kubernetes.io

[3] https://polaris-slo-cloud.github.io

3.2.1 Illustrative Scenario

In Section 2.2.1 we presented the use case of a headless CMS that is deployed as Software-as-a-Service (SaaS) and which should fulfill a high-level cost efficiency SLO.

Cost efficiency is a high-level metric that is often defined as the number of requests per second served faster than N milliseconds divided by the total cost of the workload [101, 133]. This high-level metric is not directly observable on the workload. Instead, it needs to be calculated by combining multiple low-level metrics. Doing this without tying the implementation to a specific time series DB is difficult, because each major time series DB has its own query language, e.g., Prometheus uses PromQL, InfluxDB[4] uses Flux, and Google Cloud Platform uses MQL[5]. To alleviate this problem, the Polaris middleware offers a DB-independent service for querying raw metrics. Once a high-level metric, e.g., the total cost of a workload, has been computed, it would be beneficial to reuse it in multiple SLOs, thus, we also provide a service for obtaining such high-level, composed metrics.

Before reading and evaluating metrics, an SLO needs to be configured by the customer and executed periodically to perform its evaluation. Once the SLO detects a violation, it has to be able to trigger an elasticity strategy to bring the workload back into a state, where the SLO is respected. Horizontal scaling is the most commonly used elasticity strategy today [191]. Nevertheless a customer should be able to choose from different elasticity strategies to trigger upon an SLO violation – yet, most SLOs today are tightly coupled with one specific elasticity strategy, e.g., the average CPU usage SLO in the Kubernetes HPA [168]. To support the aforementioned flexibility, the runtime's SLO control mechanism, which should be generic enough to be shared among all SLOs, must provide these features.

3.2.2 Research Challenges

RC-1 *Decoupling of SLOs from elasticity strategies*: Many SLOs are tightly coupled with the elasticity strategy they trigger. For example, HPA in Kubernetes provides an average CPU usage SLO, which can trigger only horizontal scaling. This rigid coupling reduces the flexibility of a system – re-implementing every useful elasticity strategy for every SLO controller is infeasible. Thus, a decoupling of SLOs from elasticity strategies is needed.

RC-2 *Enable realization of high-level SLOs, based on complex metrics*: Most metrics that guide cloud elasticity today are directly measurable at the system or application level [45, 240, 191]. While HPA supports custom metrics using the custom and external metrics APIs[6], both approaches require developers to write a custom API

[4] https://www.influxdata.com

[5] https://cloud.google.com/monitoring/mql/reference

[6] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-metrics-apis

server, to which the Kubernetes API can proxy requests, thus, increasing development and maintenance effort. The external metrics API supports the specification of custom queries, but this feature must also be implemented by the custom API server. An SLO middleware must provide mechanisms for combining multiple low-level metrics into high-level metrics that are reusable in multiple SLOs.

RC-3 *Cloud platform and datastore independence*: The configuration of autoscaling solutions is commonly specific to a cloud vendor or orchestrator. Likewise, there is a distinct query language for each major time series DB. Portable SLOs require mechanisms to make them independent of particular vendors.

Our first contribution, the orchestrator-independent SLO controller, addresses all three research challenges, while our second contribution, the provider-independent SLO metrics collection and processing mechanism, focuses on RC-2 and RC-3. Our third contribution, the CLI Tool, is a supporting mechanism for leveraging the other two.

3.3 Framework Overview

In this Section we provide a high-level overview of the Polaris middleware's architecture and the Polaris CLI.

3.3.1 Architecture

The architecture of the Polaris middleware is divided into two major layers, as illustrated in Figure 3.1. The *Core Runtime* layer contains orchestrator-independent abstractions and algorithms. The *Connectors* layer below it, contains orchestrator and DB-specific implementations of interfaces from the core runtime to allow connecting it to a specific orchestrator or time series DB, which are located underneath this layer. *SLO Controllers* are built on top of the core runtime, shielding them from orchestrator- and DB-specific APIs. We subsequently describe each of the runtime components in Figure 3.1 briefly:

The *Core Model* contains abstractions for defining and implementing SLOs. The most important ones are `ServiceLevelObjective`, `SloTarget`, `SloMapping`, and `ElasticityStrategy`. `ServiceLevelObjective` defines the interface that the SLO implementation needs to realize to plug into the control loop provided by the runtime. `SloTarget` is an abstraction used identify the target workload that the SLO should be applied to. An SLO is configured through an `SloMapping`, which associates a particular SLO type with a target workload and an elasticity strategy, thus, establishing a loose coupling between them. SLO mappings are deployed to the orchestrator as custom resources. Each SLO mapping type entails the definition of a custom resource type in the orchestrator. The addition of a new SLO mapping resource instance, activates the respective SLO controller, which subsequently enforces the SLO. The `ElasticityStrategy` that is specified as part of an SLO mapping, identifies the strategy that should be used if the target violates the SLO, to bring it back into a state, where the SLO is adhered. Akin to an SLO

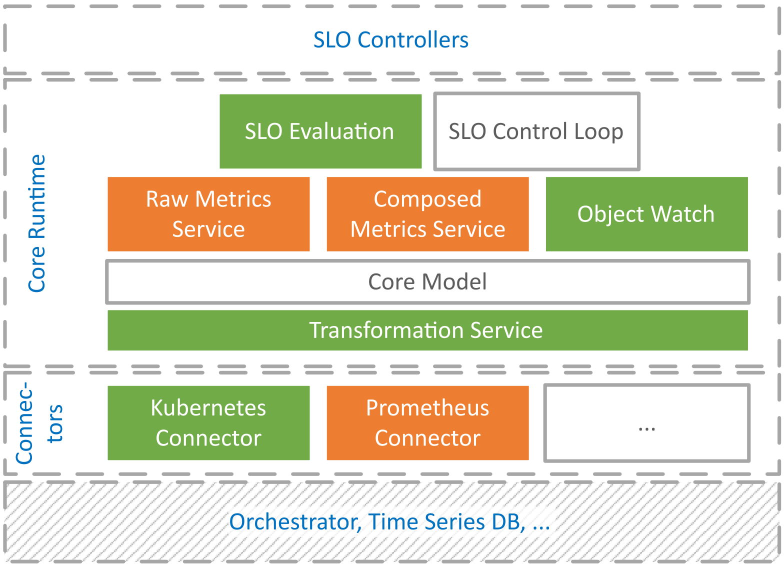


Figure 3.1: Polaris Runtime Architecture (the colors indicate, which connector realizes interfaces from a particular component).

mapping type, each `ElasticityStrategy` type is represented by a custom resource type in the orchestrator.

The *SLO control loop* is used in an SLO controller to watch the orchestrator for new or changed SLO mappings and to periodically evaluate the SLO. It relies solely on Polaris middleware abstractions and does not need be customized by an orchestrator connector or an SLO controller, albeit this is possible, if desired.

The *SLO Evaluation* facilities are used by the SLO control loop to perform the evaluation of the SLO and to trigger elasticity strategies on the orchestrator, if necessary. The evaluation of the SLO is handled by the core runtime, while the mechanisms for triggering an elasticity strategy, which are specific to each orchestrator, must be provided by the respective orchestrator connector.

The *Transformation Service* allows transforming orchestrator-independent Polaris middleware objects into orchestrator-specific objects for a particular target platform and vice-versa. The runtime provides a transformation mechanism that allows orchestrator connectors to register a type transformer for every object type that needs to be customized for the target orchestrator.

The *Object Watch* facilities allow observing a set of resource instances of a specific type in the orchestrator for additions, changes, and removals of instances. This is used, e.g., by the SLO control loop to monitor additions of or changes to SLO mappings. Orchestrator connectors must implement these facilities for their respective platforms.

The *Raw Metrics Service* enables DB-independent access to time series data to obtain metrics. A DB connector must transform the generic queries produced by this service into queries for its particular DB.

The *Composed Metrics Service* provides access to higher-level metrics, called *composed metrics*, which allow combining multiple lower level metrics into a reusable high-level metric. To make it accessible through the Composed Metrics Service, a composed metric may be packaged into a library that can be included in an SLO controller or it can be exposed as a service or stored in a shared DB, promoting loose coupling between the metrics providers and the SLO controllers. The implementation of the sharing mechanism may be provided by either the orchestrator or the DB connector.

The connectors create the bridge between the core runtime and the underlying orchestrator and DBs.

The *Kubernetes Connector* library provides Kubernetes-specific realizations of the three runtime facilities that are highlighted in green in Figure 3.1. Kubernetes-specific transformers plug into the Transformation Service to enable the transformation of objects from the core model to Kubernetes-specific objects. The library also implements the object watch facilities for the Kubernetes orchestrator, which allow the SLO control loop in an SLO controller to watch a particular SLO mapping CRD for additions of new resource instances or changes to existing ones. The SLO evaluation realization for Kubernetes augments the generic evaluation facility from the core runtime by allowing it to trigger an elasticity strategy using Kubernetes CRD instances.

The *Prometheus Connector* implements the generic Raw Metrics Service using queries specific to the Prometheus time-series database. It also supplies a mechanism for reading composed metrics from Prometheus.

3.3.2 Polaris CLI

The Polaris Command-Line Interface (CLI) provides a mechanism for project creation, building, and deployment for developers, who want to use the Polaris middleware to create custom SLOs and controllers. Its aim is to provide a convenient user interface to developers, as well as a starting point for integrating Polaris middleware projects in Continuous Integration (CI) pipelines. The major commands are the following:

`polaris-cli generate <componentType> <name>` adds a component of the specified type to the project. The `componentType` may currently be one of three types:

- `mapping-type` creates a new SLO mapping type that can be used by consumers to apply and configure an SLO.
- `slo-controller` creates an SLO controller for an SLO mapping type and deployment configuration files.
- `mapping` creates a new mapping instance for an existing SLO mapping type. This is intended to be used by consumers, who want to configure and apply a particular SLO to their workload.

`polaris-cli (docker-)build <name>` executes the build process for the specified component to produce deployable artifacts. For an SLO mapping type, this is a library package that can be published for use by customers. For SLO controllers, a container image with the executable controller for deployment on the orchestrator is produced. For an SLO mapping, the output is a configuration file, representing an instance of the corresponding SLO type CRD.

`polaris-cli deploy <name> [destination]` deploys the build artifact of the specified component to the specified destination orchestrator.

The Polaris CLI provides a default implementation for all commands, but allows developers to override these defaults in the project file, enabling, for example, the use of a different tool for deployment of the artifacts.

3.4 Mechanisms

In this Section, we describe the two main mechanisms provided by the Polaris middleware, i.e., the orchestrator-independent SLO controller and the provider-independent SLO metrics collection and processing mechanism.

3.4.1 Orchestrator-Independent SLO Controller

The central mechanism in an SLO controller is the SLO control loop – it monitors and enforces an SLO configured by a user. The overall architecture and extension points of the SLO control loop were already introduced in Section 2.4. This section will examine the details from the runtime perspective. As previously mentioned, the SLO control loop is orchestrator-independent and merely requires some supporting services to be implemented by the orchestrator connector.

The SLO control loop consists of two sub-loops, as shown in Figure 3.2. The *watch loop* on the left side (highlighted in green) is concerned with observing additions, changes, or deletions of SLO mappings in the orchestrator using the object watch facilities and maintaining the list of SLOs managed by the control loop. The *evaluation loop* on the right side (highlighted in blue) periodically evaluates each SLO and triggers the configured elasticity strategy using the SLO evaluation facilities.

Watch Loop

The watch loop begins by observing the SLO mapping custom resource types that the SLO controller supports. To this end, it uses the object watch facilities, which emit an event whenever an object of the watched types (i.e., the supported SLO mappings) is added, changed, or removed from the orchestrator. This functionality must be implemented by the orchestrator connector. Each watch event entails receiving the raw SLO mapping object that has been added, changed, or removed. Since this object is specific to the

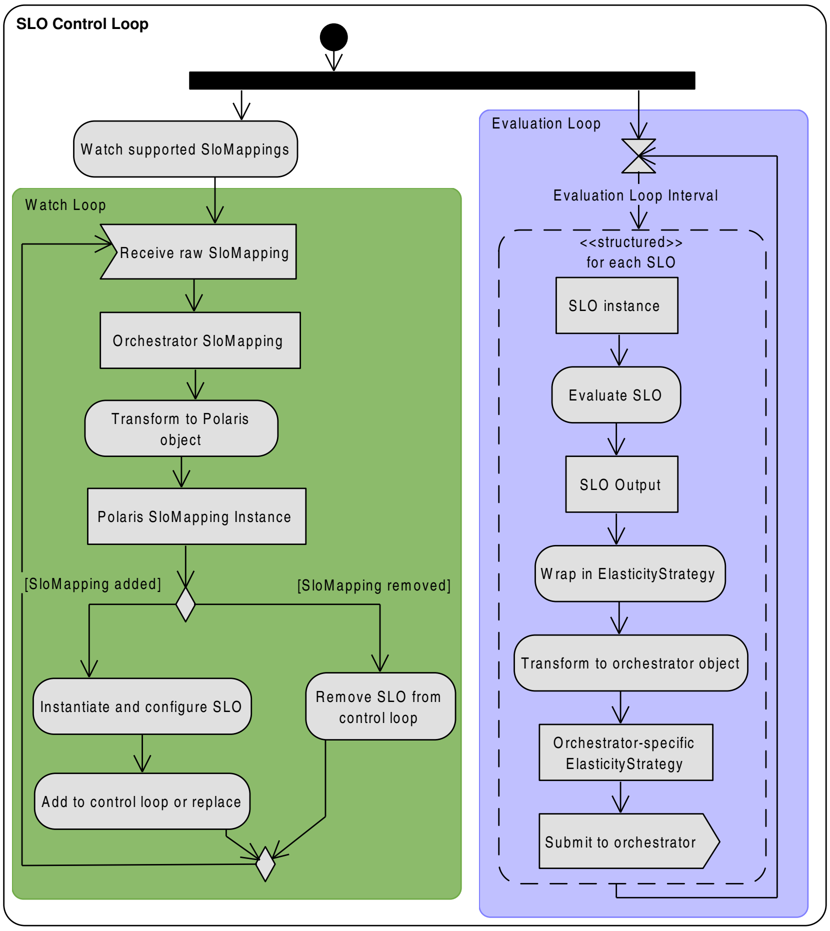

SLO Control Loop
Watch supported SloMappings
Watch Loop
Receive raw SloMapping
Orchestrator SloMapping
Transform to Polaris object
Polaris SloMapping Instance
[SloMapping added]
[SloMapping removed]
Instantiate and configure SLO
Remove SLO from control loop
Add to control loop or replace
Evaluation Loop
Evaluation Loop Interval
<<structured>>
for each SLO
SLO instance
Evaluate SLO
SLO Output
Wrap in ElasticityStrategy
Transform to orchestrator object
Orchestrator-specific ElasticityStrategy
Submit to orchestrator

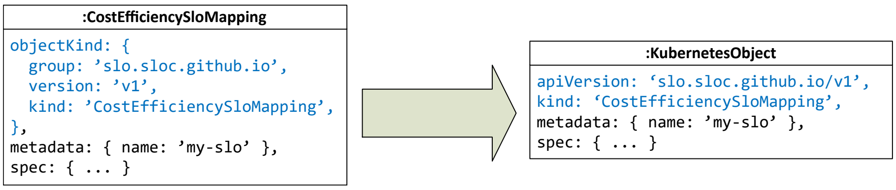
Figure 3.2: SLO Control Loop.

underlying orchestrator, it is transformed using the Transformation Service into an orchestrator-independent object.

The watch loop then acts according to the type of watch event. If a new SLO mapping has been added or changed, the appropriate SLO object that is capable of evaluating the SLO is instantiated, configured, and added to or replaced in the list of SLOs for the evaluation loop. If an existing SLO mapping has been removed, the corresponding SLO object is removed from the evaluation loop. Subsequently, the watch loop goes back to waiting for the next event.

Evaluation Loop

The evaluation loop executes at an interval that is configurable by the SLO controller. Whenever it is triggered, the evaluation loop iterates through the list of all its SLOs. For each SLO, the current status is evaluated using the SLO evaluation facilities. The exact evaluation process depends on the implementation of the particular SLO that is



```
objectKind: {
  group: 'slo.sloc.github.io',
  version: 'v1',
  kind: 'CostEfficiencySloMapping',
},
metadata: { name: 'my-slo' },
spec: { ... }
```

```
apiVersion: 'slo.sloc.github.io/v1',
kind: 'CostEfficiencySloMapping',
metadata: { name: 'my-slo' },
spec: { ... }
```

Figure 3.3: Cost efficiency SLO mapping before and after transformation.

built on top of the Polaris middleware. Generally, SLO evaluation entails the retrieval of all relevant input metrics using the Raw Metrics Service and the Composed Metrics Service. These metrics may be further processed and combined and are subsequently compared to the ideal values configured by the user in the SLO mapping. This results in an *SLO output* that indicates if the SLO is currently fulfilled, violated, or outperformed (i.e., fulfilled by a large margin, such that e.g., a resource reduction is possible) and any additional information necessary to get it back into a fulfilled state, if necessary.

This output is wrapped in an elasticity strategy object of the type specified by the SLO configuration. The elasticity strategy object is subsequently transformed to an orchestrator-specific object using the Transformation Service and submitted to the orchestrator, where it will trigger the respective controller for the elasticity strategy. This submission to the orchestrator is the part of the SLO evaluation facilities that must be implemented by the orchestrator connector – the remainder of the SLO evaluation is orchestrator-independent.

The SLO control loop is designed to handle errors during the evaluation of an SLO gracefully, such that a problematic SLO does not cause the entire controller to fail.

The SLO control loop relies on the Transformation Service to convert between orchestrator-independent and orchestrator-specific objects. All objects that are received from or submitted to the orchestrator pass through this service. Orchestrator connector libraries can register transformers for object types, whose orchestrator-specific data structure does not match that of the corresponding type in the Polaris middleware. The Transformation Service is responsible for transforming the structure of objects, while serialization and deserialization (e.g., to/from JSON) are handled by the object watch and SLO evaluation facilities. To transform an object's structure, the Transformation Service recursively iterates through all attributes of an input object. If a transformer has been registered for an attribute's type, it is executed on the attribute's value according to the direction of the current transformation operation (i.e., from orchestrator-independent to orchestrator-specific or from orchestrator-specific to orchestrator-independent). If no transformer is registered for a particular type, the value is copied and the recursive iteration continues on the value's attributes. Figure 3.3 exemplifies how an SLO mapping object for a cost efficiency SLO is transformed to a Kubernetes resource object (observe that the `objectKind` attribute of the Polaris object is transformed into two attributes, `apiVersion` and `kind`, on the Kuberenetes object).

Another essential mechanism in an SLO controller is the decoupling of SLOs and Elasticity

Strategies. The goal of this is two-fold: i) allow an SLO to trigger a user-configurable elasticity strategy that is unknown at the time the SLO controller is built (i.e., the SLO controller cannot have a hardcoded set of elasticity strategy options) and ii) allow an elasticity strategy to be used by multiple SLOs to avoid having to reimplement the same set of elasticity strategies for every SLO.

To achieve both goals, we have defined a common structure for elasticity strategy resources, consisting of three parts: a reference to the target workload, the output data from the SLO evaluation, and static configuration parameters supplied by the user. The target workload reference and the static configuration parameters are copied from the SLO mapping by the Polaris middleware. The configuration parameters are specific to the elasticity strategy that the user has chosen, which does not limit the generality of the mechanism, because they are statically specified together with the identifier of the elasticity strategy that the user has chosen and are not modified by the SLO. Conversely, the SLO evaluation output data are entirely produced by the SLO controller. The structure of the SLO output determines which elasticity strategies can be combined with that SLO, i.e., if an elasticity strategy supports the SLO's output data type as input, the two are compatible. The Polaris middleware only needs to copy the SLO output data to the elasticity strategy resource.

Using a generic data structure that is supported by multiple SLOs and elasticity strategies as an SLO's output data type, increases the number of possible SLO/elasticity strategy combinations. Any suitable data structure can be used for this purpose. The Polaris middleware includes the generic `SloCompliance` data type, which captures the compliance to an SLO as a percentage: a compliance value of 100% indicates that the SLO is exactly met, a higher value means that the SLO is violated and that, e.g., an increase in resources is needed, while a lower value indicates that the SLO is being outperformed and that resources can be reduced to save costs. To avoid too frequent scaling, `SloCompliance` includes the possibility for specifying a tolerance value, within which no elasticity action should be performed.

3.4.2 Provider-Independent SLO Metrics Collection And Processing Mechanism

The metrics required for evaluating an SLO can be obtained through two mechanisms: i) the Raw Metrics Service and ii) the Composed Metrics Service. The former is intended for low-level metrics that are directly measurable on a workload, e.g., CPU usage or network throughput, while the latter allows obtaining higher-level metrics that are aggregations of several lower-level metrics or predictions of metrics.

Raw Metrics Service

The Raw Metrics Service enables the DB-independent construction of queries for time series data. To this end, it allows specifying the metric name and the target workload, for which it should be obtained, as well as the time range and filter criteria. Furthermore, it provides arithmetic and logical operators and aggregation functions to operate on the metrics. Upon execution, a query is transformed into the native query language of the used time series DB. The result of a query is an ordered sequence or a set of ordered sequences of primarily simple (i.e., numeric or Boolean) raw or low-level metric values.

The Raw Metrics Service is designed as a fluent API [145, 79], which means that the code resulting from its use should be natural and easy to read. Specifically this entails chaining of method calls, supporting nested function calls, and relying on object scoping. Listing 3.1 shows a query for the sum of the durations of all HTTP requests that were made in the last minute, grouped by request paths.

```
rawMetricsService.getTimeSeriesSource()
  .select(’my_workload’,
    ’request_duration_seconds_count’,
    TimeRange.fromDuration(
      Duration.fromMinutes(1)))
  .filterOnLabel(LabelFilters.regex(
    ’http_controller’, ’my_workload.∗’))
  .sumByGroup(LabelGrouping.by(’path’))
  .execute();
```

Listing 3.1: Raw Metrics Service query for total duration of all HTTP requests in the last minute, grouped by paths.

Composed Metrics Service

The Composed Metrics Service is aimed at high-level metrics. These may be simple values (e.g., numbers or Booleans) or complex data structures. Unlike a raw (low-level) metric, a composed metric is not directly observable on a workload, but needs to be calculated, e.g., by aggregating several lower level metrics. A composed metric may also represent predictions of future values of a metric. Every composed metric has a *composed metric type* definition that specifies the data structure of its values and a unique name for identification.

The calculation of a composed metric requires an additional entity, termed a *composed metric source*, to perform this calculation. Each composed metric source supplies a metric of a specific composed metric type. A composed metric type is similar to an interface in object-oriented programming; it specifies the type of composed metric that is delivered and may be supplied by multiple composed metric sources.

Apart from its composed metric type, a composed metric source is also identified by the type of target workload it supports. This enables high-level metrics, such as cost efficiency, which need to be computed differently for various workload types. For example, for a REST service, cost efficiency relies on the response time of the incoming HTTP requests, a metric that is not available on a SQL database. There, the execution time of the queries could be used instead. This entails different composed metric sources, which can be registered to the respective workload types.

The Composed Metrics Service supports both, i) composed metric sources integrated into the SLO controller through libraries and ii) out-of-process composed metric sources that execute within their own metric controller. The former option computes the composed metric within the SLO controller and is simple to realize for developers, because it only requires the creation of a custom library that needs to be imported in the SLO controller and registered with the Composed Metrics Service. The latter option is more flexible and allows for decoupling of the implementation and maintenance of the SLO controller from that of the composed metric source.

Out-of-process composed metric sources may be implemented, e.g., as REST services or through the use of a shared DB. The latter allows the composed metric to be calculated once and reused by multiple SLO controllers. An out-of-process composed metric source can be leveraged to flexibly update or change the way a certain composed metric type is computed. For example, a `TotalCost` composed metric type is of interest to multiple SLOs. It may be supplied by a metric controller with a refresh rate of five minutes, i.e., the total cost of a workload is updated every five minutes. This metric controller can be replaced by a newer version, with a refresh rate of one minute, without having to recompile and redeploy the SLO controllers that depend on this composed metric.

3.5 Implementation

In this Section, we briefly describe the implementation of the mechanisms from Section 3.4 in our core runtime and the connectors for Kubernetes and Prometheus.

The Polaris middleware and its CLI are realized in TypeScript and published as a set of npm library packages. An SLO controller is a Node.js application that uses these packages as dependencies to implement SLO checking and enforcement mechanisms. All middleware and CLI code, as well as example controllers, are available as open source[7].

3.5.1 Orchestrator-Independent SLO Controller

The orchestrator-independent SLO controller relies on the abstractions provided by the core model, as well as the object watch and SLO evaluation facilities. Figure 3.4 shows the main components involved in the SLO control loop. In case the default control loop implementation does not suffice for a particular scenario, the runtime may be configured to use a custom implementation of the `SloControlLoop` interface.

The SLO control loop manages `ServiceLevelObjective` objects, which are implemented by the SLO controller. The `ObjectKindWatcher` is provided by the orchestrator connector library to enable observation of the supported SLO mapping types. The evaluation loop evaluates registered SLOs using the `SloEvaluator` provided by the orchestrator connector. The default implementation handles the evaluation of the SLO and the wrapping of its output in an elasticity strategy object – the connector library

[7] `https://polaris-slo-cloud.github.io`

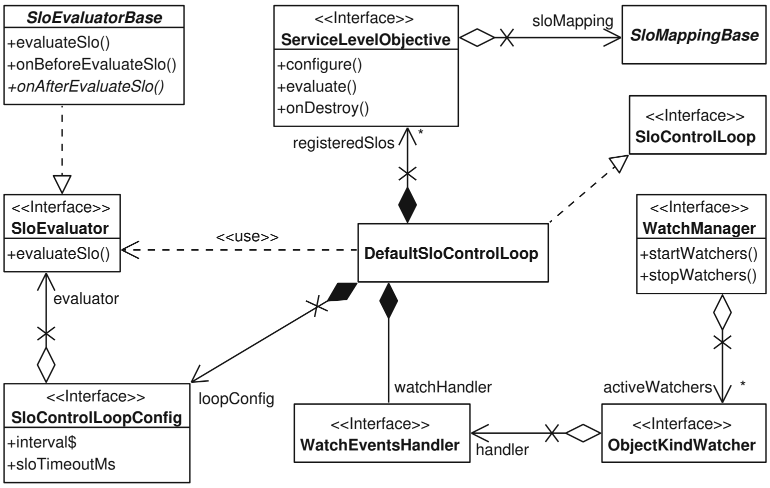


Figure 3.4: SLO Control Loop components (simplified).

must only implement the submission to the orchestrator. The Kubernetes connector for the Polaris middleware relies on kubernetes-client[8], the officially supported JavaScript client library for Kubernetes. It is important to note that the decisions need to be made inside the SLO- and elasticity strategy-specific code in the respective controllers. The purpose of the Polaris middleware is to connect an SLO to any compatible elasticity strategy and to provide reusable facilities to reduce the effort of developing these types of controllers.

The Transformation Service is relies on the open-source library class-transformer[9] for executing the transformation process, but provides its own, more flexible, transformer registration mechanism. We assume that all raw orchestrator resources contain a metadata property that uniquely identifies their type. An orchestrator connector library is required to register a transformer that converts these metadata into an `ObjectKind` object, which is the Polaris abstraction used for identifying orchestrator resource types. The Transformation Service supports associating object kinds with Polaris classes to enable the transformation into the correct runtime objects.

The decoupling of SLOs and elasticity strategies relies on a common layout of elasticity strategy resources and the use of the same data type for the output of an SLO and the input of an elasticity strategy. The user selects an elasticity strategy for an SLO by specifying its object kind in the SLO mapping that configures the SLO. After evaluating an SLO, the Polaris middleware instantiates the elasticity strategy class associated with this object kind and copies the SLO output data to it. An SLO Mapping requires the user to choose exactly one elasticity strategy. An elasticity strategy is responsible for ensuring that its sub-actions do not conflict with each other, e.g., if it combines horizontal and vertical scaling. Unlike metrics, SLOs and elasticity strategies cannot be composed.

[8] https://github.com/kubernetes-client/javascript

[9] https://github.com/typestack/class-transformer

However, it is possible to configure multiple SLOs for a single workload and, thus, also multiple elasticity strategies (one for each SLO). Since such combinations are highly use case specific, there is no generic conflict resolution mechanism. Instead, the user needs to ensure that there are no conflicts, which, however, does not limit the expressiveness of the solution.

3.5.2 Provider-Independent SLO Metrics Collection And Processing Mechanism

Raw Metrics Service

To create a raw metrics query, the Raw Metrics Service is used to obtain a `TimeSeriesSource`, which realizes a DB-independent interface for assembling time series queries for a particular target DB. The supported sources are registered with the Polaris middleware when the SLO controller starts. The `select()` method of `TimeSeriesSource` creates a new query by specifying the name of the metric and the target workload. Each method call on a query (see Listing 3.1) returns an immutable object that models the query up to this point. The query may be executed, using the `execute()` method, or extended by adding another query clause with an additional method call, which yields a new, immutable query object. This approach allows reusing a base query object, e.g., the time series of all HTTP request durations, for multiple queries without side effects, e.g., for the sum of all request durations and for the average duration of a request. When `execute()` is called on a query object q, the segments of the query chain, starting from the `select()` query object up to query object q, are passed to a `NativeQueryBuilder`. This builder needs to be provided by a DB connector library, e.g., the Prometheus connector.

Composed Metrics Service

To get a composed metric, a `ComposedMetricSource` is obtained from the Composed Metrics Service using a composed metric type and the target workload. Upon startup, the SLO controller registers all `ComposedMetricSource` realizations that are provided through libraries, together with their corresponding composed metric types and supported target workload types, in the Polaris middleware. These composed metric sources execute their metric computation logic inside the SLO controller, e.g., by using the Raw Metrics Service internally for retrieving and aggregating multiple raw metrics. If no `ComposedMetricSource` has been registered for a particular composed metric type, the Composed Metrics Service assumes that this is an out-of-process composed metric source, which is realized by a connector library. The Prometheus connector provides a `ComposedMetricSource` realization that relies on Prometheus as a shared DB, where standalone composed metric controllers store their computed metrics.

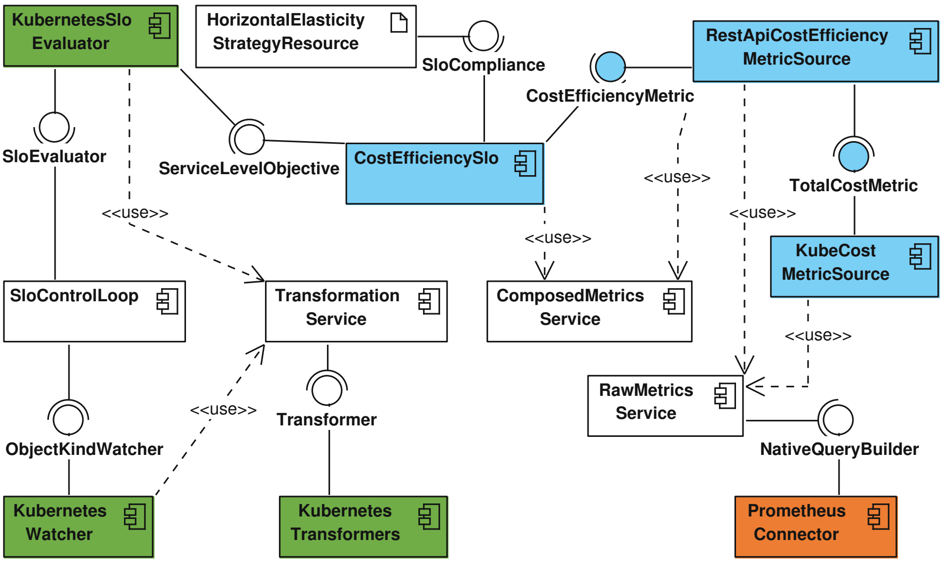


Figure 3.5: Cost Efficiency SLO Implementation (blue), Kubernetes Connector (green), Prometheus Connector (orange).

3.6 Evaluation

We implement the motivating cost efficiency SLO use case from Section 3.2 to show the productivity benefits of using the Polaris middleware and run experiments in our cluster testbed to evaluate its performance.

3.6.1 Demo Application Setup

Figure 3.5 provides an overview of the components of the cost efficiency SLO implementation and their relationships – blue components are implemented for the use case, white components are orchestrator and DB-independent parts of the Polaris middleware, and green and orange components are part of the Kubernetes and Prometheus connectors respectively. All code is available in the Polaris project's repository.

Our test cluster provides an elasticity strategy for horizontal scaling, which is also part of the Polaris project and accepts generic `SloCompliance` data as input.

First, we set up an SLO mapping type in an npm library to allow users to configure the cost efficiency SLO. To this end, the Polaris CLI can create a TypeScript class `CostEfficiencySloMapping`, which we extend with the configuration parameters. Currently, the YAML code for registering a Kubernetes CRD must be written manually or be generated from an equivalent data structure defined in Go – as part of future work, we will extend the Polaris CLI to support generating CRDs from the TypeScript SLO mapping classes.

To enable reuse of the cost efficiency metric, we implement it as a composed metric in a library. In our use case, cost efficiency is defined as the number of REST requests handled faster than N milliseconds, divided by the total cost of the workload. However, cost efficiency is not only useful for REST services, but can be applied to other types of services as well, e.g., a weather prediction service, albeit with a different raw metric as the numerator in the equation. To allow this, we define a generic cost efficiency composed metric type (composed metric types are shown as interfaces in Figure 3.5) that can be implemented by multiple composed metric sources. Thus, we enable a cost efficiency SLO controller to support multiple workload types (e.g., REST services and prediction services) by either registering multiple cost efficiency composed metric sources from libraries or by relying on out-of-process composed metric services to provide the cost efficiency metric for the various workload types. In our use case we supply a cost efficiency implementation for REST services, but since the Composed Metrics Service differentiates between workload types when obtaining a composed metric source, this can be increased to an arbitrary number of implementations for various workload types.

Since total cost is an important metric in cloud computing, this part of the cost efficiency composed metric could be reused by other composed metrics or SLOs. To this end, we create a total cost composed metric type that may be supplied by multiple composed metric sources. We provide an implementation that relies on KubeCost[10], which we use to export the hourly resource costs to Prometheus. In the implementation of the `KubeCostMetricSource`, we use the Raw Metrics Service to obtain these costs and the recent CPU and memory usage of the involved workload components and multiply and sum them to obtain the total cost.

The `RestApiCostEfficiencyMetricSource` also relies on the Raw Metrics Service to read the HTTP request metrics from the time series DB. It uses the Composed Metrics Service to obtain the cost efficiency composed metric source for the current workload to calculate the cost efficiency. The modular approach of the composed metrics allows changing parts of the implementation (e.g., use a different cost provider) without affecting the rest of the composed metrics. Note that even though we use Prometheus in our use case, the implementation of both composed metrics is completely DB-independent – in fact, a DB connector must be initialized by the SLO controller (Prometheus connector in Figure 3.5) to provide a `NativeQueryBuilder` for generating queries for a specific DB.

Next, the SLO controller needs to be created. Its bootstrapping code generated by the Polaris CLI initializes the Polaris middleware, the Kubernetes and Prometheus connectors, registers the cost efficiency SLO and its SLO mapping type with the SLO control loop, and starts the control loop. For the `CostEfficiencySlo` class, a skeleton is generated to realize the `ServiceLevelObjective` interface – it must be implemented by developers. Since the cost efficiency composed metric has been developed as a library, we need to call its initialization function during controller startup to register the cost efficiency metric with the Composed Metrics Service.

The SLO control loop monitors `CostEfficiencySloMapping` resources in the orchestra-

[10] https://www.kubecost.com

tor through the object watch facilities. To this end, the Kubernetes connector provides an implementation of the `ObjectKindWatcher` interface, which relies on the Transformation Service to transform Kubernetes resources using the transformers supplied by the Kubernetes connector as well.

When a `CostEfficiencySloMapping` resource is received by the SLO control loop, the `CostEfficiencySlo` class is instantiated to handle its evaluation, when periodically triggered by the control loop through the SLO evaluation facilities. We use the Composed Metrics Service in the `CostEfficiencySlo` class to obtain the composed metric source for the cost efficiency metric. The current value of the metric is compared to the target value configured by the user and an SLO compliance value is calculated and returned to the SLO evaluation facilities, whose orchestrator-specific parts are realized by the Kubernetes connector. They use the elasticity strategy object kind configured in the SLO mapping instance to create a `HorizontalElasticityStrategy` resource to wrap the `SloCompliance` output and submit that to the orchestrator to trigger the elasticity strategy controller.

3.6.2 Qualitative Evaluation

Due to the use of the generic `SloCompliance` (depicted as an interface in Figure 3.5) and the dynamic instantiation of the elasticity strategy resource, the cost efficiency SLO does not need to know about the specific elasticity strategy that will be used. Similarly, the horizontal elasticity strategy controller does not require any information on the SLO that has created the elasticity strategy resource. The type of SLO output data is the only link that connects an SLO to an elasticity strategy; apart from having to share the same output/input data type, they are completely decoupled. For example, changing to a vertical elasticity strategy, only entails the user altering the SLO mapping instance, used to configure the SLO, to reference a vertical elasticity strategy object kind instead of a horizontal elasticity strategy object kind.

All orchestrator-specific actions used in the SLO control loop are encapsulated in the object watch and SLO evaluation facilities, as well as the transformers used by the Transformation Service, which, in this use case, are implemented by the Kubernetes connector library. Switching to a different orchestrator, e.g., OpenStack[11], only entails exchanging the Kubernetes connector library for an OpenStack connector library (i.e., importing a different library and changing one initialization function call), the rest of the cost efficiency SLO controller's implementation would remain unchanged. The same applies to changing the type of time series DB used as the source for the raw metrics needed to compute the cost efficiency composed metric: the Prometheus connector library could be exchanged, e.g., for an InfluxDB connector library, without altering the implementation of the cost efficiency composed metric source.

Table 3.1 summarizes the line counts of the involved components. The Polaris middleware has the largest part, with 89% of the total code. The reusable total cost and the cost

[11] `https://www.openstack.org`

Table 3.1: Lines of Code (excl. comments and blanks).

Component	Lines of Code	% of Total
Composed Metrics	209	7%
SLO Controller	119	4%
Polaris Middleware	2594	89%
Total	2922	100%

efficiency metrics together add up to 209 lines or 7% of the code. The cost efficiency SLO controller is the smallest part with only 119 lines (4% of the total code), about half of which can be generated by the Polaris CLI. This shows that the usage of the Polaris middleware greatly increases productivity when developing complex SLOs, while keeping them portable to multiple orchestrators and DBs. To better illustrate the usage of the Polaris CLI, we have published a demo video online[12].

3.6.3 Performance Evaluation

Our testbed consists of a three-node Kubernetes cluster, with one control plane node and two worker nodes, all running MicroK8s[13] v1.20 (which is based on Kubernetes v1.20). The underlying virtual machines (VMs) are running Debian Linux 10 and have the following configurations:

- *Control plane & Worker1*: 4 vCPUs and 16 GB of RAM
- *Worker2*: 8 vCPUs and 32 GB of RAM

We use a synthetic workload for the performance tests, as this is the best practice for stress tests. To the best of our knowledge, there is no other middleware that offers the same features as Polaris. Out of the production-ready solutions, HPA offers the greatest similarity. However, the realization of composed metrics would require the addition of a custom Kubernetes API server to provide these metrics, which means that it could not compete with Polaris with respect to the lines of code. We conduct two experiments, where we create 100 cost efficiency SLO mappings and let the SLO controller evaluate them at an interval of 20 seconds.

SLO Controller Resource Usage

First, we show that an SLO controller built with the Polaris middleware does not consume excessive resources, even when handling numerous SLOs. For this experiment, we deploy the cost efficiency SLO controller to our cluster in a pod with resource limits of 1 vCPU and 512 MiB RAM. We observe the resource usage over a period of 20 minutes using

[12] https://www.youtube.com/watch?v=3_z2koGTExw

[13] https://microk8s.io

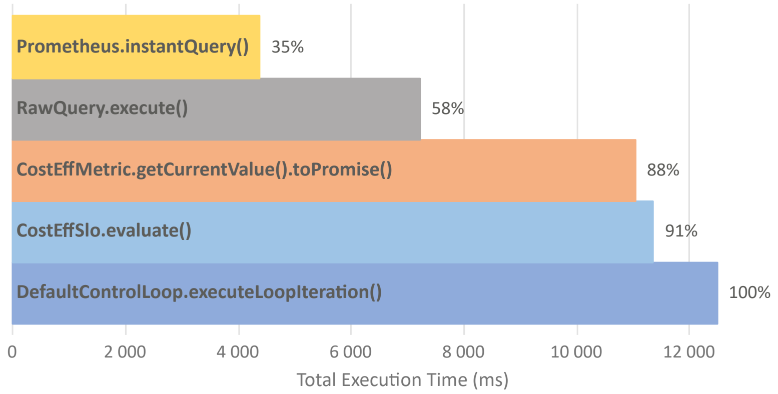


Figure 3.6: Average total execution times of `executeControlLoopIteration()` and its children across all 300 seconds profiling sessions.

Grafana[14], which fetches metrics from Prometheus. While evaluating 100 SLOs every 20 seconds, the CPU usage stays between 0.2 and 0.25 vCPUs, while the memory usage is between 102 and 140 MiB. Thus, both, CPU and memory usage stay far below the pod's limits and constitute reasonable values for execution in the cloud.

Execution Performance of the Polaris Middleware

Next, we demonstrate that the Polaris middleware does not add significant overhead to an SLO controller. To this end, we execute the cost efficiency SLO controller on a development machine (Intel Core i7 Whiskey Lake-U with 4 CPU cores, clocked at 1.8 GHz and 16 GiB RAM) under the Visual Studio Code JavaScript debugger and profiler, while being connected to our cluster's control plane node through SSH. As for the previous experiment, we use 100 cost efficiency SLO mappings to generate load. We execute 3 profiling sessions, each with a length of 300 seconds (i.e., 5 minutes).

Figure 3.6 shows a flame chart with the total execution times of all SLO control loop iterations and the major methods invoked by it. The numbers are the mean average values across all profiling sessions. The sum of the execution times of all SLO control loop iterations in a 300 second profiling session is on average 12,480 milliseconds (ms). The SLO control loop itself and the triggering of elasticity strategies using the results from the SLO evaluations only takes about 9% of that time, the remaining 91% are consumed by the evaluation of the cost efficiency SLO. The SLO relies on the cost efficiency composed metric, which takes up most of the SLO's execution time. The composed metric sets up one raw metrics query itself for the HTTP request metrics and delegates the creation of the query for the costs to the total cost composed metric. The execution of both raw metrics queries amounts to about 58% of the total SLO control loop execution time. More than half of this (35% of the total) amounts to the query execution in the third-party

[14] `https://grafana.com`

Prometheus client library. This analysis demonstrates that the evaluation of SLOs using the Polaris middleware is performant and does not show any evidence of bottlenecks.

3.7 Summary

In this chapter, we presented the Polaris middleware, a flexible middleware system for implementing complex metrics and SLOs that trigger elasticity strategies in an orchestrator- and DB-independent manner. We have motivated the need for the Polaris middleware using a real-world use case of REST service that needs to fulfill a high-level cost efficiency SLO and listed its architecture requirements.

We presented the design and implementation of the mechanisms that enable our core contributions:

1. The orchestrator-independent SLO controller provides the runtime for periodically evaluating SLOs and triggering elasticity strategies. The control loop and mechanisms for triggering the elasticity strategy are completely provided by Polaris middleware, such that developers can focus on the business logic of their SLO.
2. The provider-independent SLO metrics collection and processing mechanism allows obtaining raw, low-level metrics from a time series DB and aggregating them into higher-level composed metrics. These composed metrics can be reused by multiple SLO controllers and they serve as an abstraction that can be realized by multiple providers, which makes it possible to switch the underlying computation of a composed metric without having to modify the SLO controller.
3. The Polaris CLI Tool allows creating and managing projects that rely on the Polaris middleware, as well as the generation of scaffolding code for SLOs, elasticity strategies, and controllers.

Finally, we used the realization of the motivating use case using the Polaris middleware to evaluate the performance of our middleware and to show that it provides substantial benefits and flexibility when implementing SLOs.

CHAPTER 4

Pogonip: Scheduling Asynchronous Applications on the Edge

Scheduling long-lived microservice-based applications in the Edge-Cloud continuum requires the consideration of network QoS properties to ensure that the application can fulfill its SLOs. Asynchronous applications that rely on a message broker for communication differ greatly in their communication patterns compared to synchronous applications. The Pogonip Scheduler is specifically designed for SLO-aware scheduling of asynchronous microservice-based applications under heterogeneous network conditions.

4.1 Introduction

Today's trend for designing efficient and scalable applications relies on the microservice architectural style [160]. It decomposes an application into autonomous and decoupled services, each with specific and independent functionalities. They are typically deployed separately to one another, especially resorting on software containers, which enable grouping a microservice with all its dependencies thus simplifying its deployment. Two main communication styles between microservices exist: synchronous and asynchronous. With synchronous communication, a microservice m_1 contacts a microservice m_2 directly and maintains the connection until it receives a response. This increases coupling and limits the application's flexibility to change. So, this is often considered an anti-pattern [51]. With asynchronous communication, m_1 and m_2 communicate through an

This chapter is based on the paper T. Pusztai, F. Rossi, and S. Dustdar, "Pogonip: Scheduling Asynchronous Applications on the Edge," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

indirection layer, e.g., a message queue [171]. This allows decoupling the application's microservices, improving scalability and flexibility.

In the last years, we are witnessing the diffusion of computing resources located at the edge of the network. Edge computing extends cloud computing by using computational capabilities of devices at the edge of the Internet [207]. This concept suits applications whose data are generated and consumed at the network periphery [2], but it brings new challenges, mainly due to the heterogeneity and decentralized distribution of computing and networking resources. In this context, the *placement (or scheduling) problem* is of utmost importance; it defines the mapping between the application microservices and the computing nodes. An improper node selection can negatively impact the application performance and, in a pay-per-use scenario, can lead to higher execution costs.

To simplify the deployment of microservice-based applications, we resort to orchestration tools like Kubernetes[1]. When a new application should be executed, Kubernetes uses a component, i.e., the *scheduler*, to solve the placement problem. Although Kubernetes is one of the most popular production-grade orchestrators [112], it has been originally designed for cluster environments, so it is not well suited to run applications on the edge. It does not consider that edge nodes may be connected with different link types (e.g., WiFi, 4G, 5G) that exhibit varying quality of service characteristics, such as latency [25]. Edge computing requires new placement strategies that explicitly take into account the presence of heterogeneous resources and non-negligible network delays. As surveyed in [24, 27], different solutions exist in literature. However, to the best of our knowledge, all of them consider only the placement of synchronous applications. Asynchronous applications exhibit peculiar features, like increased throughput, that cannot be neglected [52]. For example, in a smart mobility scenario, where cars and road-side devices report traffic and safety data to the analyzer microservices, a message queue significantly reduces coupling between the participating microservices allowing a non-blocking communication. In such applications, the queue represents a key component that should be allocated as close as possible to all microservices, as it is the logically centralized communication component.

In this chapter, we present *Pogonip*, an edge-aware scheduler for Kubernetes, designed for asynchronous microservice-based applications. The main contributions are as follows:

1. We formulate the placement problem as an Integer Linear Programming (ILP) optimization problem. It places microservices by considering constraints on network latency towards the queue system on edge nodes. Moreover, if not enough edge resources are available, it can offload microservices to third-party cloud nodes, while keeping the additional costs low.

2. We define the Pogonip heuristic to quickly find an approximate solution for real-world execution scenarios, because the optimization problem is NP-hard.

[1] `https://kubernetes.io`

3. We implement the heuristic as a scheduler prototype for Kubernetes and release it as open-source.

Using an asynchronous edge application and a Kubernetes cluster, we evaluate our solutions against two state of the art placement policies and the default Kubernetes scheduler.

The remainder of this chapter is structured as follows. We first discuss our system model and the problem to solve (Section 4.2). Then, we formulate the optimization placement problem (Section 4.3) and the Pogonip heuristic (Section 4.4). In Section 4.5, we describe the heuristic integration in Kubernetes and, in Section 4.6, we present the experimental evaluation. Section 4.7 concludes the chapter.

4.2 System Model and Problem Definition

In the following, we focus on identifying edge-aware placement solutions for asynchronous microservice-based applications. We consider an edge-cloud environment shared by multiple independent applications. For each application, we assume that its microservices are highly decoupled and that they communicate through a message queue. In Table 4.1, we summarize the used notations.

We consider a geographically distributed edge environment, where multiple edge clusters provide computing resources on-demand. The edge resources are organized in different edge clusters. An edge cluster can be modeled as a graph $G = (N, E)$, where the set of nodes N represents the distributed computing resources and the set of links E represents the logical connectivity between nodes. We characterize each edge node $n \in N$ with the following attributes: C_n, the available computing resources in n; M_n, the available memory in n; P_n, the cost (on a time basis) of using n for hosting application components. We characterize each logical link $(n, m) \in E$ with the network latency $d_{n,m}$ between the nodes n and m. Such a logical connectivity between computing resources results from the underlying physical network paths and routing strategies. These attributes can be known a-priori or can be monitored and estimated at run-time. Each edge cluster has a control node (CN), the entry point of the cluster. When a client submits an application to the CN, the edge cluster scheduler solves the placement problem. We denote as $\mathbb{A}$ the set of all managed asynchronous applications. An application $A \in \mathbb{A}$ consists of multiple microservices and a queue system q. We define $i \in A$ as an application component, i.e., a microservice instance or the queue system. We assume that the user correctly sizes the queue system q, so that it can sustain the application workload without affecting application integrity and performance. To simplify the problem formulation, we use $A' = A \backslash \{q\}$ when the queue system should not be considered. Each application component i is characterized by the required CPU C_i and memory M_i. Differently from synchronous applications, asynchronous applications usually do not aim to minimize response time, because microservices indirectly interact with one another. In a distributed environment, we are interested in allocating microservices close to the message queue, so that they can quickly receive messages from the queue. Therefore, each application $A \in \mathbb{A}$ exposes its requirements in terms of $ND_{A,\max}$, i.e., the maximum network delay between the queue and each microservice allocated on edge resources. For allocating the application components, the edge cluster scheduler can select nodes from the edge or from the cloud. The key idea is to first grant edge resources and then the cloud ones, if there are not enough computing resources on the edge. We denote as A^* the microservices forwarded to the cloud. In general, propagating any application component to the cloud introduces costs and communication delays, which can be detrimental for the application performance. For our investigation, we can reasonably assume that the cloud offers almost infinite computing capacity. We denote as S the set of cloud nodes. We characterize each cloud node $s \in S$ with its available CPU capacity, C_s, memory capacity, M_s, and cost, P_s. We consider cloud resources that are managed by a third party, so we should favor the utilization of edge resources. This is the case of the queue system that, being the application's key communication component, should be up and running for all the application life time. Conversely, the application microservices can be managed more easily, as they can be restarted on a different location without compromising the application availability (so we can temporarily place them on cloud resources). For this reason, we assume that queue systems can be placed only on edge nodes.

Following a *divide et impera* approach, we divide the placement problem formulation in two sub-problems, i.e., edge and cloud placement problem. This simplifies the placement problem formulation, speeding up the resolution phase and allowing to more easily integrate other objective goals. The *edge placement problem* takes into account the placement on the current edge cluster. If the edge nodes do not have enough resources, some of the application microservices are forwarded to the cloud for processing. In this case, we should solve a second problem, i.e., the *cloud placement problem.* The goal of this problem is to minimize the cost of the used cloud resources, which are rented from a third party.

4.3 Optimization Problem Formulation

In this section, we formulate optimization problems to solve the edge and cloud placement problems of asynchronous microservice-based applications.

4.3.1 Edge Placement

We model the application placement in the edge cluster with binary variables $x^A_{i,n}$, $A \in \mathbb{A}$, $i \in A$, $n \in N$, where $x^A_{i,n} = 1$ if the component i of the application A is placed on the edge node n, and $x^A_{i,n} = 0$, otherwise. For each $A \in \mathbb{A}$, we use the binary variables z^A_i, with $i \in A$, to indicate the application components to execute in the cloud: $z^A_i = 1$ if the application component i is forwarded to the cloud and $z^A_i = 0$ otherwise. We denote the application placement on edge resources with the vector $\boldsymbol{x} = \langle x^A_{i,n} \rangle$, with $A \in \mathbb{A}$, $i \in A$, and $n \in N$ and the application components forwarded to the cloud with the vector $\boldsymbol{z} = \langle z^A_i \rangle$, with $A \in \mathbb{A}$, $i \in A$.

Table 4.1: Placement Problem Notations.

Entity	*Notation*	*Definition*
Edge Cluster	CN	Control Node of the edge cluster
	$\mathbb{A}$	Set of applications to place
	N	Set of nodes within the edge cluster
	C_n	CPU capacity of node $n \in N$
	M_n	Memory capacity of node $n \in N$
	P_n	Cost of node $n \in N$
	$d_{n,m}$	Network latency between nodes $n \in N$ and $m \in N$
Cloud	A^*	Set of microservices forwarded to the cloud
	S	Set of cloud nodes
	C_s	CPU capacity of node $s \in S$
	M_s	Memory capacity of node $s \in S$
	P_s	Cost of node $s \in S$
Application	$A = \{q\} \cup A'$	Set of application components, with $A \in \mathbb{A}$
	q	Message queue system of application A
	A'	Set of Application Microservices
	C_i	CPU demand of the application component $i \in A$
	M_i	Memory demand of the application component $i \in A$
	$ND_{A,max}$	Maximum network latency required by A

Edge Resources Cost. For any application component placed on edge nodes, we incur a resource cost, $F(\boldsymbol{x})$:

$$F(\boldsymbol{x}) = \sum_{n \in N} P_n \cdot f_n \tag{4.1}$$

where the binary variables f_n denote whether $n \in N$ hosts at least one component (i.e., a microservice instance or a message queue system). Therefore we define $f_n, \forall n \in N$, as follows:

$$\frac{\sum_{A \in \mathbb{A}} \sum_{i \in A} x_{i,n}^{A} + \zeta_n}{\Gamma} \leq f_n \leq \sum_{A \in \mathbb{A}} \sum_{i \in A} x_{i,n}^{A} + \zeta_n \tag{4.2}$$

where Γ is a large number and ζ_n is a constant. $\zeta_n = 1$ if n already hosts at least one application component, 0 otherwise. Note that, if $P_n = 1$, the edge resources cost counts only the number of edge nodes used for running the applications.

Cost of Forwarding to Cloud. An edge cluster aims to run microservices locally, optimizing resource utilization of edge nodes. However, to correctly deploy the application, the CN has to enforce the allocation of all of the application microservices. To extend edge resources, the CN can use the cloud. This results in a cost of forwarding microservices to the cloud $Z(\boldsymbol{z})$, which we assume to be proportional to the number of forwarded microservices' instances:

$$Z(\boldsymbol{z}) = \sum_{A \in \mathbb{A}} \sum_{i \in A'} z_i^A \tag{4.3}$$

Application Constraints. Considering application-level requirements, the placement policy explicitly models the network delay between nodes, and allocates the application microservices only on nodes n and $v \in N$ whose network delay $d_{n,v}$ is below an application-defined critical value $ND_{A,\max}$. Note that microservices communicate asynchronously through the message queue. Therefore, for each application $A \in \mathbb{A}$, it is important that each microservice $i \in A'$ is as close as possible to the queue system q. We define $\gamma_{i,q}^A$ as the network distance between the microservice i and the queue system of the application A. Formally, $\forall A \in \mathbb{A}$ and $\forall i \in A'$ we have:

$$\gamma_{i,q}^A = \sum_{(n,v) \in N \times N} y_{(i,q)(n,v)}^A \cdot d_{n,v} \tag{4.4}$$

The $y_{(i,q)(n,v)}^A$ variables model the logical AND between placement variables $x_{i,n}^A$ and $x_{q,n}^A$, $\forall A \in \mathbb{A}, i \in A \backslash \{q\}$ and $n, v \in N$: $y_{(i,q)(n,v)}^A = x_{i,n}^A \cdot x_{q,v}^A$.

Similarly, for each $A \in \mathbb{A}$, also the queue system q should be as close as possible to the CN, being the CN the access point to the edge cluster. Therefore, we formalize the following constraint: $d_{CN,n} \cdot x_{q,n}^A \leq ND_{A,\max}$.

Edge Placement Problem Formulation. We formulate the placement problem as an ILP model that determines the optimal mapping between the applications' components and the edge nodes. Our problem formulation considers an objective function that minimizes the edge and cloud resources cost. We define the objective function $G(\boldsymbol{x}, \boldsymbol{z})$ as the sum of the QoS metrics to be minimized:

$$G(\boldsymbol{x}, \boldsymbol{z}) = F(\boldsymbol{x}) + Z(\boldsymbol{z}) \tag{4.5}$$

The Edge Placement problem is formulated as follows:

$$\min_{\boldsymbol{x},\boldsymbol{z}} G(\boldsymbol{x}, \boldsymbol{z})$$

$$\textbf{subject to:}$$

$$\sum_{n \in N} x_{i,n}^A + z_i^A = 1, \quad \forall A \in \mathbb{A}, \forall i \in A \tag{4.6}$$

$$z_q^A = 0, \quad \forall A \in \mathbb{A} \tag{4.7}$$

$$\sum_{A \in \mathbb{A}} \sum_{i \in A} C_i \cdot x_{i,n}^A \leq C_n, \quad \forall n \in N \tag{4.8}$$

$$\sum_{A \in \mathbb{A}} \sum_{i \in A} M_i \cdot x_{i,n}^A \leq M_n, \quad \forall n \in N \tag{4.9}$$

$$d_{CN,n} \cdot x_{q,n}^A \leq ND_{A,\max}, \quad \forall A \in \mathbb{A}, n \in N \tag{4.10}$$

$$\gamma^A_{i,q} \leq ND_{A,\max}, \quad \forall A \in \mathbb{A}, i \in A' \tag{4.11}$$

$$\sum_{v \in N} y^A_{(i,q)(u,v)} = x^A_{i,u}, \ \ \forall A \in \mathbb{A}, i \in A', u \in N \tag{4.12}$$

$$\sum_{u \in N} y^A_{(i,q)(u,v)} \leq x^A_{q,v}, \ \forall A \in \mathbb{A}, i \in A', v \in N \tag{4.13}$$

$$\frac{1}{\Gamma}\left(\sum_{A \in \mathbb{A}} \sum_{i \in A} x^A_{i,n} + \zeta_n\right) \leq f_n \quad \forall n \in N \tag{4.14}$$

$$\sum_{A \in \mathbb{A}} \sum_{i \in A} x^A_{i,n} + \zeta_n \geq f_n \quad \forall n \in N \tag{4.15}$$

$$x^A_{i,n} \in \{0,1\} \quad \forall n \in N, A \in \mathbb{A}, i \in A \tag{4.16}$$

$$z^A_i \in \{0,1\} \quad \forall A \in \mathbb{A}, i \in A \tag{4.17}$$

$$f_n \in \{0,1\} \quad \forall n \in N \tag{4.18}$$

where (4.6) ensures that all the application components are either assigned to edge nodes or forwarded to the cloud, and guarantees that each of them is placed on one and only one node. The constraint (4.7) forces the placement of the queue systems on edge nodes only. Constraints (4.8) and (4.9) limit the placement of the application components on an edge node $n \in N$ according to its available resources, while (4.10) and (4.11) limit the network delays among edge nodes used to run the application. Constraints (4.12) and (4.13) model the logical AND between the placement variables. Finally, (4.14) and (4.15) define the f_n variables, $\forall n \in N$, indicating whether n is used to run any of the application components.

4.3.2 Cloud Placement

If any microservice instance should be forwarded to the cloud, we have to solve the *cloud placement problem*. For each microservice $i \in A^*$, we model the microservice i placement on a cloud node $s \in S$ with new binary variables $t_{i,s}$: where $t_{i,s} = 1$ if microservice i is placed on the cloud node s and $t_{i,s} = 0$ otherwise. We denote the cloud placement vector as $\boldsymbol{t} = \langle t_{i,s} \rangle$, with $i \in A^*$ and $s \in S$.

Cloud Resources Cost. The cloud resources cost $P(\boldsymbol{t})$ accounts for the active cloud nodes for running the applications' microservices:

$$P(\boldsymbol{t}) = \sum_{s \in S} \delta_s \cdot P_s \tag{4.19}$$

where the binary variables δ_s denote whether $s \in S$ is active and hosts at least one microservice. We formally define $\delta_s, \forall s \in S$ as follows:

$$\frac{\sum_{i \in A^*} t_{i,s} + \psi_s}{\Gamma} \leq \delta_s \leq \sum_{i \in A^*} t_{i,s} + \psi_s \tag{4.20}$$

where Γ is a large number and ψ_s is a constant such that $\psi_s = 1$ if s hosts at least a microservice (as result of previous optimization rounds), 0 otherwise.

Cloud Placement Problem Formulation. We formulate the cloud placement problem as an ILP problem which defines a mapping of the applications' microservices on the cloud nodes with the aim of minimizing the cost of used cloud resources. The Cloud Placement problem is formulated as follows:

$$\min_{\boldsymbol{t}} P(\boldsymbol{t}) \tag{4.21}$$

$$\sum_{s \in S} t_{i,s} = 1 \qquad \forall i \in A^* \tag{4.22}$$

$$\sum_{i \in A^*} C_i \cdot t_{i,s} \leq C_s \qquad \forall s \in S \tag{4.23}$$

$$\sum_{i \in A^*} M_i \cdot t_{i,s} \leq M_s \qquad \forall s \in S \tag{4.24}$$

$$\frac{\sum_{i \in A^*} t_{i,s} + \psi_s}{\Gamma} \leq \delta_s \qquad \forall s \in S \tag{4.25}$$

$$\sum_{i \in A^*} t_{i,s} + \psi_s \geq \delta_s \qquad \forall s \in S \tag{4.26}$$

$$t_{i,s} \in \{0,1\} \qquad \forall s \in S, \forall i \in A^* \tag{4.27}$$

where (4.22) ensures that all the forwarded microservices are placed on cloud nodes. The constraints (4.23) and (4.24) limit the placement of microservices on a cloud node $s \in S$ according to its available resources. Finally, (4.25)–(4.27) are used to define the δ_s variables, $\forall s \in S$.

4.4 The Pogonip Heuristic

Allocating asynchronous applications on (edge or cloud) computing resources is an NP-hard problem; so, the ILP formulations might not scale well as the problem instance increases in size. To overcome this issue, we propose the Pogonip greedy heuristics. First, we present the *greedy edge placement heuristic*, a network-aware policy to determine the placement of asynchronous applications in the edge cluster. Then, we describe the *greedy cloud placement heuristic* that allows to reduce the number of cloud nodes used to host the forwarded application microservices.

4.4.1 Greedy Edge Placement Heuristic

The proposed greedy edge placement heuristic solves a variant of the bin-packing problem, while taking into account the available computing resources and the network delays between edge nodes (see Algorithm 4.1). First, it sorts the unplaced applications by their $ND_{A,\max}$ requirement in ascending order, i.e., the first applications of the list have more stringent $ND_{A,\max}$ values (line 4). Then, it places one application at a time (line 6–8). For each application A, the heuristic identifies N^q, the set of edge nodes that can host the queue system q and that have a network delay to the CN below $ND_{A,\max}$ (lines 11–12). If N^q is empty, the application is discarded. Otherwise, the heuristic computes β_n for each

Algorithm 4.1 Placement Heuristic on Edge Nodes

```
 1: Input:  $\mathbb{A}$: Applications to deploy; $N$: Set of edge nodes;
 2: Output:  $A^*$: Microservices forwarded to cloud;
 3: Output:  $X$: Application placement;
 4: $\mathbb{A}$ = Sort $A \in \mathbb{A}$ by $ND_{A,\max}$ (in ascending order)
 5: $X = \{\}$
 6: for all $A \in \mathbb{A}$ do
 7:     applicationPlacement($A$, $N$, $X$, $A^*$)
 8: end for

 9: function APPLICATIONPLACEMENT($A$, $N$, $X$, $A^*$)
10:     $q \leftarrow$ Queue system of application $A$
11:     $N^q \leftarrow$ Filter $n \in N$ on $q$ resource requirements
12:     $N^q \leftarrow$ Filter $n \in N^q$ on $d_{CN,n} \leq ND_{A,\max}$
13:     if $N^q$ is empty then
14:         discard application $A$
15:         return
16:     end if
17:     Compute $\beta_n = \min(\lfloor \frac{C_n - C_q}{C_q} \rfloor, \lfloor \frac{M_n - M_q}{M_q} \rfloor), \forall n \in N^q$
18:     $x^A_{q,n_q} \leftarrow$ Allocate $q$ on $n_q$ having maximum value of $\beta_n$
                                                   ▷ spread applications across nodes
19:     $X \leftarrow X \cup x^A_{q,n_q}$
20:     for all microservice $i \in A \backslash \{q\}$ do
21:         $N^i \leftarrow$ Filter $n \in N$ on $i$ resource requirements
22:         $N^i \leftarrow$ Filter $n \in N^i$ on $d_{n_q,n} \leq ND_{A,\max}$
23:         if $N^i$ is empty then
24:             $A^* \leftarrow A^* \cup i$
25:             continue;
26:         end if
27:         Compute $\beta_n = \min(\lfloor \frac{C_n - C_i}{C_i} \rfloor, \lfloor \frac{M_n - M_i}{M_i} \rfloor), \forall n \in N^i$
28:         $x^A_{i,n_i} \leftarrow$ Allocate $i$ on $n_i$ having minimum value of $\beta_n$
                                                   ▷ maximize resource utilization
29:         $X \leftarrow X \cup x^A_{i,n_i}$
30:     end for
31: end function
```

$n \in N^q$. The β_n factor estimates the node n capacity of hosting q, approximating the number of q instances that can be executed on n, considering the most critical resource (line 17). The edge node having maximum value of β_n is selected for the allocation of q. This allows to spread queue systems across nodes, avoiding node congestion. Similarly, for each microservice i, the heuristic identifies the edge nodes N^i (lines 21–22). If N^i is empty, i is forwarded to the cloud. Otherwise, the heuristic greedily chooses the first candidate node that minimizes β_n (line 27). Note that this avoids spreading microservices across the computing nodes, while preferring to minimize the number of active nodes.

4.4.2 Greedy Cloud Placement Heuristic

The cloud placement heuristic uses a greedy approach to place the microservices received from edge control nodes (see Algorithm 4.2). For each microservice $i \in A^*$ and cloud node

Algorithm 4.2 Placement Heuristic on Cloud Resources

```
1: Input:  $A^*$: Microservices to deploy; $S$: Set of cloud nodes;
2: Output:  $T$: Application placement;
3: for all microservice $i \in A^*$ do
4:     $S^i \leftarrow$ Filter $s \in S$ on $i$ resource requirements
5:     Compute $\beta_s = \min(\lfloor \frac{C_s - C_i}{C_i} \rfloor, \lfloor \frac{M_s - M_i}{M_i} \rfloor)$, $\forall s \in S^i$
6:     $t_{i,s_i} \leftarrow$ Allocate $i$ on $s_i$ having minimum value of $\beta_s$
7:     $T \leftarrow T \cup t_{i,s_i}$
8: end for
```

$s \in S$, the heuristic filters cloud resources according to the resource requirements of i, expressed in terms of CPU C_i and memory M_i demand. First, the heuristic computes β_s, which estimates the node s capacity of hosting i (line 5). The cloud node with minimum β_s value is selected to host i. This allows to reduce the number of used cloud nodes and, as a consequence, cloud usage cost. At the end, the application placement T is accordingly updated.

4.5 Prototype

In this section, we present the prototype implementation of our heuristic, realized as a Kubernetes scheduler.

4.5.1 Kubernetes Scheduler

A pod is the smallest deployment unit in Kubernetes. It consists of one or more tightly coupled containers that are co-located and scaled as an atomic entity. Each application component (i.e., a microservice or the queue system) is deployed using a pod. Kubernetes ensures that a given number of pods are up and running using a *Replica Set*. To manage the deployment of applications, the *Deployment* object is built upon the Replica Set concept, exposing a higher level abstraction, simplifying the pods' update and providing additional functionality (e.g., rolling updates). To manage stateful applications, whose pods must be deployed in a particular order, have persistent IDs, and/or always be connected to the same storage volumes (e.g., RabbitMQ, which is used in Section 4.6), Kubernetes introduces the *Stateful Set* concept. Differently from Deployments, a Stateful Set maintains a sticky identity for each managed pod, allowing its state recovery. When a new pod is created, Kubernetes triggers the scheduler to identify a suitable hosting node. The default Kubernetes scheduler is *kube-scheduler*, which is implemented using the *scheduling framework* [237], an extensible architecture for Kubernetes schedulers. It decomposes the scheduling process into two cycles: scheduling and binding. From a high-level perspective, the *scheduling cycle* is a sequence of filtering and scoring stages. First, it identifies the nodes that can run the pod by applying a set of filters. Then, it assigns a score to all eligible nodes according to different criteria. Finally, it selects the node with the highest score to host the pod. If multiple nodes achieve the same score, one of them is randomly selected. The mapping between the pod and the chosen

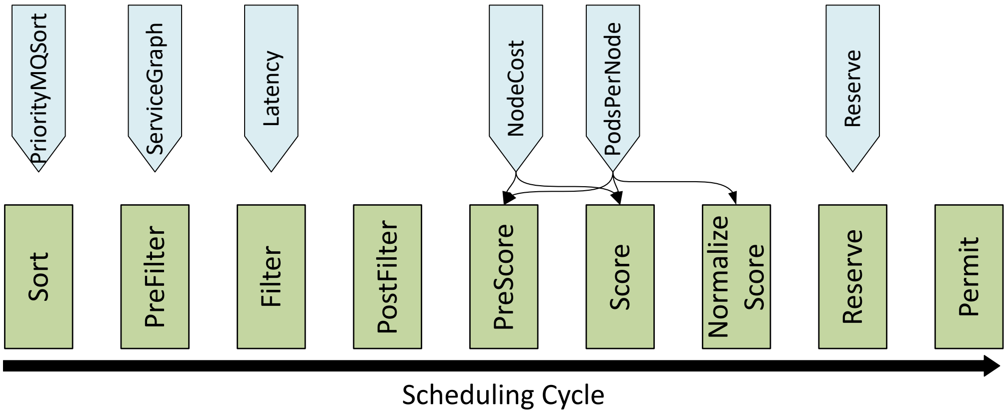


Figure 4.1: Scheduling cycle (adapted from [237]) and Pogonip plugins.

node is committed to the cluster by the *binding cycle* [237]. The kube-scheduler includes a placement policy that spreads pods on computing resources located in the cluster. As such, it is not well-suited for placing pods in an edge computing environment and dealing with its heterogeneity. However, the modularity of Kubernetes allows us to easily integrate custom placement policies. There are two main ways to customize the placement process: (1) by changing the configuration of the default scheduler; or (2) by implementing a custom scheduler that runs instead of the default one [200]. Changing the configuration of the default scheduler is limited by the capabilities of kube-scheduler, requiring a custom scheduler for more advanced customizations. A custom scheduler is any application that observes the list of pods and assigns pods to nodes. However, relying on the modular architecture of the scheduling framework simplifies the development of new placement policies, splitting their logic into multiple and decoupled stages. Additionally, the scheduling framework allows reusing some or all the plugins from kube-scheduler [235], such as, e.g., `NodeResourcesFit`, which filters out nodes that do not satisfy a pod's resource requirements. Kubernetes schedules an application's pods independently: a failure to schedule/place one pod has no effect on the scheduling status of the other pods. The scheduler will retry to place a failed pod later. When the scheduling framework's sequence of stages cannot be applied (e.g., as in the optimal placement formulation), an independent custom scheduler is required.

4.5.2 Prototype Architecture and Implementation

The Pogonip scheduler implements the greedy heuristics presented in Section 4.4 leveraging the scheduling framework of Kubernetes v1.20.1 and most of its default plugins. The prototype is published as open-source under the name *rainbow-scheduler* in the RAINBOW[2] project's orchestration package[3].

[2] https://rainbow-h2020.eu
[3] https://gitlab.com/rainbow-project1/rainbow-orchestration

Pogonip needs to be able to identify the pods that contain a queue system and be aware of the application's maximum tolerable network delay. For this prototype, we use Kubernetes labels to attach this information to each pod. As part of future work within RAINBOW, a service graph abstraction will be developed that will contain this and other relevant information about an application. Pogonip augments the default kube-scheduler functionality by adding the plugins to the scheduling cycle shown in Figure 4.1. The green boxes show the scheduling cycle stages that are executed for every pod. Each stage provides an extension point, for which plugins can be registered. The `Sort` stage determines the order in which the incoming pods will be handled (only one plugin can be active in this stage). `PreFilter` and `Filter` are responsible for filtering out nodes that cannot host the newly added pod, e.g., because they have too few resources. The `PreFilter` stage is executed per pod to prepare information needed in the `Filter` stage, which, conversely, is executed for every node. If the set of remaining nodes is empty, the `PostFilter` plugins are executed. `PreScore` and `Score` plugins assign a score to each eligible node. Similarly to the filter-related stages, the `PreScore` stage is executed once per pod, while the `Score` stage is executed once per node. A `NormalizeScore` extension may be registered for each `Score` plugin to normalize its scores as an integer between 0 and 100, as is required by the scheduling framework. After this stage, the node with the highest score is selected to host the pod. `Reserve` plugins are notified with the outcome, allowing to update third-party data structures. `Permit` plugins are executed in the last stage of the scheduling cycle, to approve, deny, or delay a pod from being admitted to the binding cycle. In the default configuration, kube-scheduler registers multiple plugins in the scheduling cycle, such as `NodeResourcesBalancedAllocation` and `NodeResourcesLeastAllocated`. While the first plugin favors nodes that would obtain a more balanced resource usage, the latter prefers nodes that have few allocated resources. Consequently, the default scheduling strategy spreads pods: it prioritizes nodes with the least number of pods, without considering their heterogeneity or geographic distribution [235].

Pogonip extends kube-scheduler with custom plugins, as shown in Fig. 4.1. Building on top of the default kube-scheduler `PrioritySort` plugin, `PriorityMqSort` prioritizes the pods belonging to applications with more stringent $ND_{A,\max}$ requirements and ensures that the queue system pods are placed before the others. `ServiceGraph` is a `PreFilter` plugin that retrieves the graph for the application that the pod is part of. The `Latency` plugin is a `Filter` plugin that removes all nodes that do not meet the application's $ND_{A,\max}$ requirement. If the pod hosts a queue system, the network latency between the current node and the edge CN is considered. Otherwise, the plugin filters nodes limiting the network latency to the node hosting the application queue system. For cloud nodes, the `Latency` plugin is disabled. The `PodsPerNode` plugin ties into the `PreScore` and `Score` extension points. First, in the `PreScore` stage, the plugin retrieves the pod's required resources. Then, in the `Score` stage, for each edge/cloud node n, it computes the β_n factor (see Section 4.4). For all application pods (but the queue system), we want to select the node n with the minimum β_n value. Once all β_n have been computed, they are normalized in the $[0, 100]$ range. Furthermore, to implement the preference for edge nodes, the PodsPerNode returns a score of zero for all cloud nodes, if at least one eligible edge node has been found. NodeCost is a Score plugin that assigns higher scores to cheaper nodes. To avoid compromising the optimizations by the Pogonip Score plugins, we disable the scheduling framework plugins NodeResourcesBalancedAllocation and NodeResourcesLeastAllocated. Finally, Reserve runs in the Reserve stage and updates the application placement.

To solve the optimal ILP placement formulation within Kubernetes, we develop a *custom scheduler* and a *Placement Resolver*. The custom scheduler is deployed as a pod and invoked by Kubernetes as soon as pods need to be allocated on the nodes. To solve the placement problem, the custom scheduler interacts with the Placement Resolver. It is an external service that exposes the ILP placement problem resolution as a service, through RESTful APIs. As soon as the custom scheduler obtains the pods placement, it defines the pod-to-node mapping using the Kubernetes abstractions.

4.5.3 Benchmark Placement Policies

In this section, we present the existing placement policies against which we evaluate our edge-aware solutions. Together with the default *kube-scheduler* policy, we include two well-known placement policies, that are often adopted in computing frameworks, namely Greedy First-fit and Round-robin.

We implement these placement policies using the Kubernetes scheduling framework. They both leverage the default PreFilter and Filter plugins to determine which nodes are capable of hosting a pod. However, we replace all default PreScore and Score plugins with a single Score plugin, which implements the corresponding placement policies.

The Greedy First-fit Heuristic. The Greedy First-fit heuristic is one of the most popular solutions used to solve the bin packing problem. It considers the application's pods as elements to be (greedily) allocated in bins, representing computing nodes. Specifically, for each pod, the Greedy First-fit policy defines the placement on the first node that fulfills the pod's resource requirements. Our GreedyFirstFit plugin greedily selects the first fitting node from the iteration order provided by the scheduling framework.

Round-robin Heuristic. The Round-robin heuristic organizes the nodes in a circular list, registering the latest node used for placement. A new pod to be allocated is assigned to the first node with enough resources, starting from the current position on the circular list. Akin to our Greedy First-fit implementation, we implement the Round-robin selection with a single RoundRobin plugin for the Score stage.

4.6 Experimental Results

We define two sets of experiments aimed to show the benefits of our placement policies when the managed application is deployed in an edge computing environment using Kubernetes. First, in Section 4.6.2, we analyze the advantages of using edge-aware policies in a heterogeneous environment. Then, in Section 4.6.3, we generalize the achieved results and show the benefits of combining edge and cloud computing resources when multiple applications should be executed. We compare the edge-aware policies, presented in Section 4.3 and 4.4, against the benchmark placement policies, presented in Section 4.5.3.

4.6.1 Experiment Setup

As reference application, we use a modified version of an IoT taxi application[4] written for the Fogify fog emulator [225, 224]. It uses real-world taxi and limousine data[5] to generate its workload. We have modified this application's microservices to communicate asynchronously through a RabbitMQ[6] queue system. A single application deployment consists of one RabbitMQ instance and four taxi app microservices: an *IoT load generator* that sends location data from the dataset once per second to two *edge aggregator* instances; the latter buffer the received data for one minute and then send them to a *data storage* microservice, which permanently stores the data. The resource requirements are the defaults used by the RabbitMQ Kubernetes Operator [23] and reasonable values for the microservices, given their purposes:

- RabbitMQ: 2 CPU cores, 2 GiB memory
- IoT load generator: 0.25 CPU cores, 0.25 GiB memory
- Edge aggregator (2x): 0.5 CPU cores, 0.5 GiB memory
- Data storage microservice: 1 CPU core, 1 GiB memory

The application requires $ND_{A,\max}$ to be equal to 50 ms.

We set up a cluster using the `kind`[7] tool, which allows running a Kubernetes cluster (v1.20.1) inside Docker, with each node being a container. Since nodes would report the CPU and memory capacity of the host VM as their available resources, we created two extended resources, `fake-cpu` and `fake-memory`, which we can explicitly configure for each node. The cluster runs on a VM with 22 virtual CPU cores and 62.9 GiB of RAM. The hosting server has an Intel Xeon CPU (Cascade Lake) with a base clock of 2.1 GHz. The cluster topology differs between the two experimental scenarios, as shown in Figure 4.2. The vertices represent edge nodes, while links denote the network connections between nodes; on each link, we report the network latency expressed in ms [196]. Both scenarios consider a single edge cluster.

As placement policies, we consider the optimal ILP formulation (referred to as OPT), the Pogonip heuristic, the Greedy First-fit heuristic, the Round-robin heuristic, and the

[4] https://github.com/UCY-LINC-LAB/fogify-demo
[5] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[6] https://www.rabbitmq.com
[7] https://kind.sigs.k8s.io

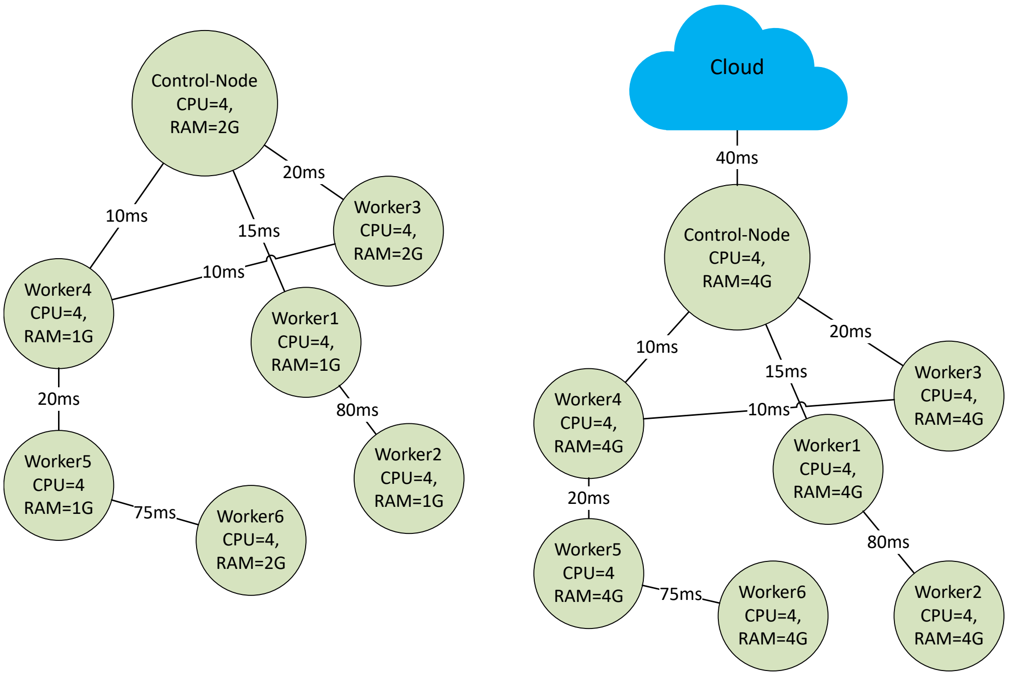


Figure 4.2: Cluster topologies in experiments: Scenario 1 (left), Scenario 2 (right).

default kube-scheduler. To solve the optimal ILP formulation, we use CPLEX 12.8. To minimize the number of used edge nodes, in the OPT we set $P_n = 1$ for each node of the edge cluster (see Eq. 4.1). For each scenario we execute 5 runs with every placement policy. All source code, including the experimental scripts, is available in our public repository.

4.6.2 Application Deployment and Network Latencies

In this experiment, we consider a single taxi application instance and only edge nodes. To model the edge environment, we have configured the `fake-cpu` and `fake-memory` resources to match those of the Raspberry Pi 3 Model B+ (4 CPU cores and 1 GiB of RAM) and the Raspberry Pi 4 Model B (4 CPU cores and 2 GiB of RAM)[8].

The goal of this experiment is to evaluate the latency between the microservices and the queue system that can be achieved by the various schedulers. Figure 4.3 summarizes the results. The different schedulers obtain very different placements for the application, including solutions where microservices are allocated far away from the message queue. In such a case, the application performance can be significantly reduced.

The Greedy First-fit policy does not meet the $ND_{A,\max}$ requirement for more than half of the microservices, with a median latency of 57.5 ms and a mean average of 56.25 ms. For each pod, it chooses the first node that fulfills the resource requirements in the

[8] `https://www.raspberrypi.org`

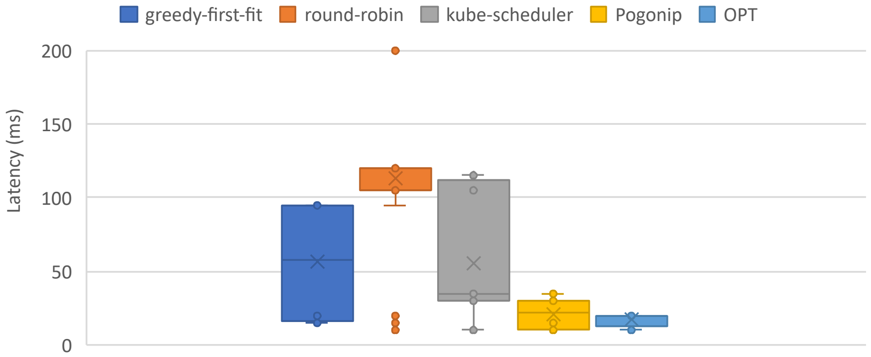


Figure 4.3: Latency between microservices and the queue system, when a single application is deployed using different placement policies.

list (`control-node`, `worker1`, ..., `worker6`). Thus, the placement is the same on every run. Four nodes are used: the queue system is placed on the `control-node` and the four application microservices as follows: one on `worker1`, two on `worker2`, and one on `worker3`. Both, `worker1` and `worker3` have low network latency towards the `control-node`, which explains Greedy First-fit's minimum values of 15 ms. However, since `worker2` has a latency of 95 ms to the `control-node`, the $ND_{A,\max}$ requirement was clearly violated.

The Round-robin policy and kube-scheduler policy spread the application pods on 5 edge nodes, which is the highest number of nodes with respect to the other configurations. The Round-robin policy organizes the edge nodes into a circular list. The queue system is placed on the `control-node` and the application microservices in the nodes `worker1` through `worker4` in the first run. This violates the $ND_{A,\max}$ requirement, because of the use of `worker2`. Subsequent runs performed even worse, because the next node in the circular list at the start of these runs was either `worker5` or `worker4`. Both are too small to host the queue system, resulting in it being placed on `worker6`, which has the second highest latency to all other nodes, thus, explaining the poor performance of the Round-robin policy.

Kube-scheduler registers a mean average latency of 56 ms and a median latency of 35 ms. Conversely to Greedy First-fit and Round-robin, kube-scheduler uses all `worker` nodes across the runs (five in every run), but avoids the `control-node`. This may be related to the fact that this node hosts the Kubernetes master.

Unlike the benchmark heuristics, OPT and Pogonip consider network delays while computing the application placement. OPT always computes the best placement, obtaining a mean average network latency of 17.5 ms and a median of 20 ms by using the `control-node`, `worker3`, and `worker4`.

With Pogonip, we register a maximum latency of 35 ms, an average of 21.5 ms, and a median of 22.5 ms. Across all runs, Pogonip uses all nodes, except for `worker2` and `worker6`, registering a slight increase in the network latencies compared to the OPT

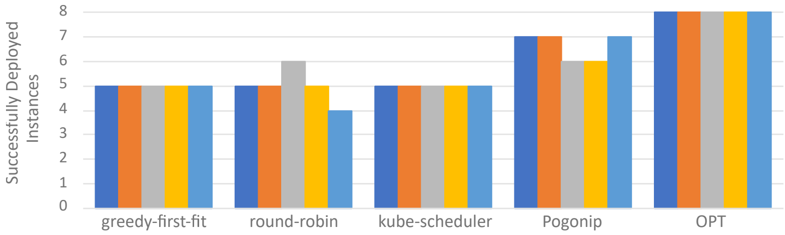


Figure 4.4: Successfully deployed application instances (out of 8 submitted) when different placement policies are used. Each experiment is run 5 times.

solution. Anyway, Pogonip always meets $ND_{A,\max}$, outperforming all previous benchmark policies.

4.6.3 Allocating Applications on Edge and Cloud Nodes

In this experiment, we consider an edge cluster that receives application deployment requests and uses its control node to allocate them for execution. We submit 8 instances of the taxi application to the edge cluster. The benchmark policies (i.e., Greedy First-fit, Round-robin, and kube-scheduler) are not designed to distinguish between edge and cloud nodes. Therefore, they cannot complement the edge with resources rented from the cloud. Conversely, our policies can benefit from the cloud. We consider the computing infrastructure depicted in Figure 4.2. All edge nodes have the resources of a Raspberry Pi 4 Model B with a hardware configuration of 4 CPU cores and 4 GiB of RAM. We use three types of cloud nodes, with 10 instances each, characterized as follows:

- *small*: 4 CPU cores, 4 GiB of RAM, cost of \$2/hour;
- *medium*: 8 CPU cores, 8 GiB of RAM, cost of \$4/hour;
- *large*: 16 CPU cores, 16 GiB of RAM, cost of \$8/hour.

Although these cloud node costs are fictional, their ratios match those of real-world cloud providers (e.g., [149]).

As soon as the pod placement is computed, Kubernetes enacts it. If not enough resources are available, some pods cannot be successfully placed, meaning that they remain in a pending state until resources are freed. Figure 4.4 shows the number of successfully deployed applications and Figure 4.5 the distribution of network latencies between the microservices and the queue system for the successfully deployed applications.

The Greedy First-fit policy successfully executes 5 application instances out of the 8 submitted, in each run of the experiment. Greedy First-fit results in network latencies to the queue system comparable to those of the previous experiment. However, in this case, the minimum latency was 0 ms; due to the more powerful nodes, some microservices are

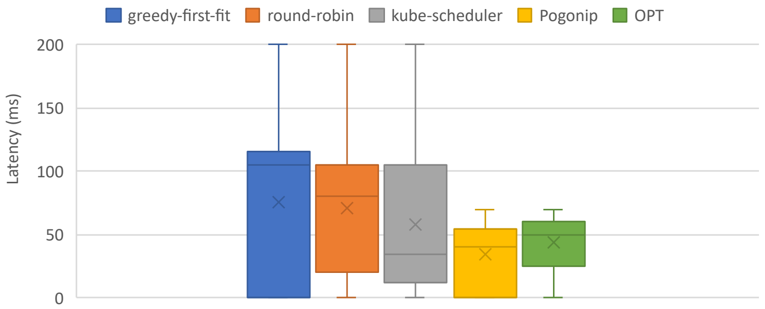


Figure 4.5: Latency between microservices and the queue system, when multiple applications are deployed using different placement policies.

co-located with their queue system. In spite of that, 58% of the application microservices exceed the $ND_{A,\max}$ constraint.

The Round-robin policy successfully executes between 4 and 6 application instances, with 5 being the median number. With a median latency of 80 ms, the $ND_{A,\max}$ constraint was violated by 56% of the microservices. As expected, the Round-robin placement is pseudo random, as it depends on the number and order of edge nodes as well as the number of pods already allocated.

Since Kubernetes does not treat the application as a whole, we observe a high variation in the number of successfully executed applications across the runs. For example, in the third run, Round-robin successfully executes 6 applications (even though 3 queue system pods are placed on nodes interconnected with high latency). Instead, in the last run, only 4 applications are successfully executed. This is due to Kubernetes, which executes one message queue pod more than in the other runs and, because of the limited available resources, this prevents other microservices from running. This reveals a non-deterministic behavior of Kubernetes in prioritizing pods for execution. First, when priorities are not explicitly assigned, pods are sorted by their creation time (see `PrioritySort` [235]). Second, the different Kubernetes controllers, e.g., Deployment and StatefulSet, work in parallel, so they concurrently manage the creation and execution of different application pods. In our case, each queue system is a StatefulSet, whereas the other microservices are controlled by a Deployment.

Kube-scheduler successfully executes 5 application instances in all runs. The latency distribution between the queue system and each microservice is slightly better than the one obtained using Round-robin. With a median latency of 35 ms, 60% of the pods fulfilled the $ND_{A,\max}$ constraint.

Pogonip executes between 6 and 7 instances in this experiment, because, unlike the previous policies, it can place the application microservices in the cloud as well (only the queue system is required to be on an edge node). With 18 to 20 pods on the edge, Pogonip places fewer pods there than the benchmark policies. This is due to the prioritization of the queue system pods, which have the highest resource requirements of all pods.

Since bigger pods are placed on the edge, fewer resources are left there for the other microservices, which are instead placed in the cloud. All application microservices are placed on three small cloud nodes, resulting in a total cloud cost of \$6/hour. This is the lowest possible value; e.g., in the first run, all pods placed in the cloud require a total of 12 GiB of memory, which could also be met by two medium nodes or one large node, both would result in the higher price of \$8/hour.

Despite placing about half of the pods on cloud nodes, Pogonip achieves much better latencies than the benchmark policies by avoiding the two high-latency edge nodes. With a median latency of 40 ms, Pogonip fulfilled the $ND_{A,\max}$ constraint for about 62% of the pods. The other pods violated the constraint by an average of 10.3 ms, i.e., less than a sixth of the next best benchmark policy, Round-robin.

Why does Pogonip not place all instances, despite prioritizing the queue system and using the cloud? Their creation timestamps showed that some queue system pods are created after some other microservice pods. We also noticed that Kubernetes starts scheduling before all pods have been created by the StatefulSet and Deployment controllers. Since pods are the schedulable units, Kubernetes considers a pod ready for scheduling as soon as it has been created. Pogonip's prioritization algorithm also has to adhere to this limitation.

OPT executes all 8 application instances, placing 20 pods on the edge and 20 pods in the cloud. It also results in the lowest latency distribution of all other policies, with a median of 50 ms. Differently from the other scheduling policies, OPT waits for all 40 pods to be created before computing the optimal placement solution for all the applications. This results in all applications being successfully executed on both edge and cloud nodes. OPT uses 4 small cloud nodes for running 20 application microservices, resulting in a cost of \$8/hour. It requires one more cloud node than Pogonip, because OPT executes one more queue system on the edge where there are fewer resources available for other microservices. The 20 pods selected for scheduling in the cloud require a total of 13.5 GiB of memory, which could be met by four small nodes, two medium nodes, or one large node – all for the same price of \$8/hour, making the selected solution the cheapest.

These two experiments showed the importance of considering application and computing features while determining the placement. By considering network latency, Pogonip and OPT reduce the communication delay between the application queue and its microservices (resulting also in limited latency variance). This can be critical to the proper functioning of an edge application. The ability of combining edge and cloud computing allows allocating a greater number of applications with respect to the other benchmark placement policies. We conclude the section with some consideration on the resolution time of each placement policy. We measure the resolution time as the time needed to compute the application placement. Technically, we measure it as the time a pod needs to move from the `PreFilter` stage until the `Reserve` stage and then sum the times for all pods in the run. For OPT, we measure it as the time needed to compute the placement for the edge and for the cloud. We conducted an additional experiment, with 10 application instances deployed at once, which constitutes a 25% increase in the number of pods that need to

Table 4.2: Scheduler Resolution Times

Scheduler	**Time per Instance (8 Total)**	**Time per Instance (10 Total)**	**Increase**
Greedy First-fit	21.6 ms	30.2 ms	40%
Round-robin	26.0 ms	32.8 ms	26%
kube-scheduler	38.4 ms	43.9 ms	14%
Pogonip	131.0 ms	152.7 ms	17%
OPT	334.8 ms	576.0 ms	72%

be placed. Table 4.2 shows execution times for a single instance and the increases in execution time between 8 and 10 instances.

Greedy First-fit and Round-robin have similar resolution times; their implementation is rather simple and they differ only in one `Score` plugin. Kube-scheduler results in a slightly longer resolution time, because it contains more `Score` plugins. Pogonip takes 131 ms on average for 8 instances and 152.7 ms for 10 instances. This is about 3.5 times as long as kube-scheduler, which is likely caused by the `Latency` plugin, which has to evaluate paths through the cluster graph. The execution time increases by about 17% between 8 and 10 instances, which is below the increment in number of pods. OPT aims to find the optimal solution of the ILP formulation; it registers the longest resolution time, with 335 ms to place 8 instances and 576 ms to place 10 instances. In this case, the resolution time increases by about 72%, which is almost three times the increment in the number of pods. We observe that even though the number of applications is rather limited, OPT requires more than half a second to find a placement solution. Of course, we expect this time to exponentially increase as the number of applications increases as well, thus resulting in an impractical approach when it comes to working in a dynamic edge environment. This limitation of OPT justifies the adoption of edge-aware placement heuristics that can compute the placement of asynchronous components more quickly.

4.7 Summary

Microservices are an architectural style for developing an application as a suite of autonomous and decoupled services, that communicate using synchronous or asynchronous techniques. Although the placement problem is widely explored in the context of synchronous applications, so far, to the best of our knowledge, the problem of allocating asynchronous applications has not been investigated.

Therefore, in this chapter, we presented an approach for solving the placement problem for asynchronous microservice-based applications in an edge environment. First, we formulate the problem as an ILP model. Since the problem is NP-hard, it may suffer from scalability issues when the number of managed microservices increases. Thus, we propose Pogonip, a novel edge-aware heuristic. It can quickly allocate asynchronous microservices by explicitly taking into account the peculiarities of edge nodes. Moreover, if microservices require more capacity than available in the edge, it can complement the computing environment by exploiting cloud computing. Integrating these policies in Kubernetes, we conducted an extensive evaluation using an edge application that processes taxi location data. The experimental results showed the benefits of combining edge and cloud resources as well as the importance of explicitly considering the edge environment's peculiarities while allocating applications, resulting in better adherence to their requirements.

CHAPTER 5

Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge

Scheduling synchronous long-lived microservice-based applications in the Edge-Cloud continuum is more challenging than the placement of asynchronous applications that was discussed in the previous chapter. Synchronous applications may exhibit complex dependencies among each other resulting from their communication patterns. Polaris Scheduler captures these dependencies using a Service Graph and provides an SLO- and network topology-aware scheduling framework for placing microservices on Cloud or Edge nodes. The framework is designed for extensibility using plugins. Polaris Scheduler includes plugins to enforce network SLOs at the time of scheduling and select nodes that are likely to maintain favorable network QoS characteristics over a long time.

5.1 Introduction

One of the biggest challenges in scheduling the microservices of a large-scale Edge application is selecting nodes that allow the application to fulfill its network SLOs. As discussed in the previous chapter, the heterogeneity of network links within an Edge cluster may lead to a node being unsuitable for hosting a microservice, despite having sufficient resources, only because its network connection is unstable. To allow an application to fulfill its SLOs, the scheduling process must not only consider the nodes' resources, but also the network when determining an optimal placement. Solving this problem

This chapter is based on the paper T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, “Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge,” in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022.

for synchronous applications is more challenging than for asynchronous applications, because the communication patterns can be much more complex if microservices can communicate directly with each other.

In this chapter we present Polaris Scheduler[1], an SLO-aware scheduler for the Edge. Our main contributions include:

1. an *SLO- and topology-aware scheduling framework*,
2. a *Service Graph and a Cluster Topology Graph* to model application SLO- and Edge network-topologies, and
3. a *suite of scheduling plugins* that leverage these abstractions and mechanisms to enforce the network SLOs at the time of scheduling.

This chapter is structured as follows: Section 5.2 outlines a realistic Edge Computing use case with strict network QoS requirements to motivate our work, Section 5.3 presents an overview of Polaris Scheduler and its scheduling pipeline, and Section 5.4 describes the components that make it SLO-aware. In Section 5.5 we evaluate our work using experiments, based on our motivating use case and Section 5.6 summarizes the chapter.

5.2 Motivation

Polaris Scheduler is part of the Polaris SLO Cloud[2] project, a SIG of the Linux Foundation Centaurus project[3], a novel open-source platform for building unified and highly scalable public or private distributed Cloud and Edge systems. Polaris aims to make SLOs first class entities in Cloud and Edge Computing [184, 183]. Polaris Scheduler builds upon our vision of broad-range SLO-awareness in Edge scheduling [165], extending it with algorithms for enforcing network QoS, as well as concrete realizations and evaluations of previously presented concepts.

5.2.1 Illustrative Scenario

To better illustrate the need for the Polaris Scheduler, we present an Edge computing use case for analyzing road traffic conditions to reveal congestion and for detecting hazards on the road to alert nearby smart cars to improve traffic safety. The traffic conditions analysis is inspired by traffic info crowdsourcing in Google Maps [49], while the hazard detection is adapted from use case 2 of RAINBOW[4] [193], a European Union Horizon 2020 Fog Computing research project. Our use case features the five main microservices, depicted in Figure 5.1. The *Collector* service receives events from nearby

[1] https://github.com/polaris-slo-cloud/polaris-scheduler/tree/v0.2.2
[2] https://polaris-slo-cloud.github.io
[3] https://www.centaurusinfra.io
[4] https://rainbow-h2020.eu

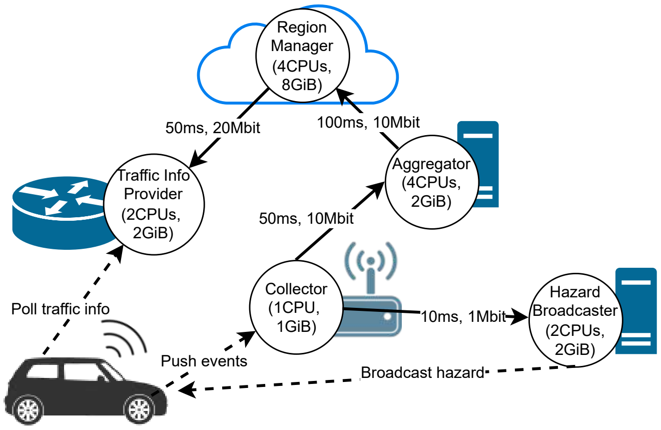


Figure 5.1: Traffic Analysis & Hazard Detection Service Graph (simplified).

cars about their movements, performs initial filtering, and detects if there is a hazard, e.g., an accident or an animal on the road. To ensure low latencies, the Collector is deployed on 5G base station nodes. The filtered data are forwarded to the Aggregator service and hazards to the Hazard Broadcaster service. The *Aggregator* service, which combines traffic and hazard data from multiple Collectors and forwards them to the Region Manager service, requires a more powerful node, e.g., a Cloudlet or an Edge gateway. The *Hazard Broadcaster* service receives hazard alerts from a Collector, determines within which vicinity vehicles need to be alerted and, subsequently, notifies them via 5G. The *Region Manager* service aggregates traffic and hazard data from all Aggregator services in the region into a unified traffic view – it needs to run in the Cloud. The unified traffic view is periodically forwarded to the *Traffic Info Provider* service instances, which allow cars to periodically pull updates to this view.

The relationships between the microservices in Figure 5.1 are annotated with network SLOs for the respective communication links. For example, the link from the Collector to the Aggregator requires a connection with a minimum bandwidth of 10 Mbps, to allow streaming the filtered event data. The maximum latency of this link is 50 ms, because the Aggregator only provides information for the unified traffic view, whereas the maximum latency from the Collector to the Hazard Broadcaster is 10 ms to ensure that detected hazards are broadcast in time to nearby vehicles. The maximum latency for collision warnings, as defined by the ETSI TS 101 539-3 standard [68] is 300 ms. In a similar use case the detection of a pedestrian using a camera was reported as taking 90-100 ms [31]. If we assume 100 ms for the detection by a smart car, another 30 ms for transmission to the Collector, 50 ms of processing by the Collector, and 20 ms by the Hazard Broadcaster, then the 10 ms SLO we have defined for the connection between Collector and Hazard Broadcaster is reasonable to allow for spare time to broadcast the alert to nearby vehicles. While the unified traffic view also contains information on hazards, these are intended for more distant cars, which means that the latency requirements are less stringent on this network path. The network SLOs are important for the user experience (unified traffic view) and absolutely critical to the safety of nearby vehicles (Hazard Broadcaster). Thus, a scheduler must ensure that the microservices' placement fulfills these SLOs.

5.2.2 Research Challenges

RC-1 *Capturing dependencies and enforcing SLOs among application microservices:* Most production schedulers, such as the Kubernetes default scheduler, place each microservice of an application completely independently of the others, ignoring their interdependencies. However, Edge applications need to be treated as a whole; dependencies and network SLOs among microservices must be considered during scheduling to allow the application to fulfill its purpose.

RC-2 *Guaranteeing long-time compliance to network SLOs*: Fulfilling an application's SLOs immediately after scheduling its microservices is the foundation for its success. However, frequent SLO violations and the associated scaling or migration of microservices may introduce unnecessary costs. Furthermore, if not addressed, this issue may cause a microservice to repeatedly be migrated back and forth within a set of nodes, because the QoS of their network connections keeps oscillating. Thus, it needs to be investigated how current and historical information about network connections can be used to infer a node's suitability to fulfill a microservice's SLOs.

RC-3 *Capturing the Edge cluster's topology and network QoS state:* Finding solutions to RC-1 and RC-2 requires information about the current topology and network QoS state of the cluster. Cloud schedulers typically assume a flat network structure, where each node is directly connected to every other node. This is often not the case in an Edge cluster, because certain nodes may be connected to a larger network through gateway nodes that may become a bottleneck. Furthermore, Edge clusters can be highly volatile: nodes may leave unexpectedly because they lose connectivity or their battery is drained or a 5G node's bandwidth may vary with the number of active devices in its cell. Representing the cluster's network state and leveraging it for scheduling is imperative for fulfilling SLOs.

5.3 Approach Overview and Scheduling Pipeline

Polaris Scheduler aims to augment the resource-based scheduling approach taken by many Cloud and Edge schedulers today with awareness of application SLOs, especially those related to network QoS. Specifically, our approach is to find a suitable placement for microservices in an Edge cluster that (i) respects the resource requirements of the workload, e.g., virtual CPU cores (vCPUs), memory, GPS, camera, and (ii) fulfills the microservices' network QoS requirements in terms of bandwidth, latency, latency variance (i.e., jitter), and packet loss, thus, allowing them to meet their network-related SLOs. To this end, we consider the interactions between the microservices of an application, i.e., the application's topology. Furthermore, Polaris Scheduler allows adding extensions to optimize the placement for additional SLOs in the future. The need to make placement decisions based on multiple requirements makes this a *multi-criteria decision making (MCDM) problem*. To enable extensibility Polaris Scheduler utilizes a plugin-based approach, where each criterion in the MCDM problem is handled by one plugin. Polaris Scheduler leverages the Edge cluster's network topology modeled as a *Cluster Topology Graph* and the interdependencies and SLOs of the microservices of an application modeled as a *Service Graph* to determine if hosting a microservice on a particular node would fulfill the network SLOs.

5.3.1 Scheduling Pipeline

Each microservice instance is a container, which needs to traverse the *scheduling pipeline* to be assigned to a node for execution. Polaris Scheduler's scheduling pipeline is based on the Kubernetes *scheduling framework* [237], which is also used by *kube-scheduler*, the default Kubernetes scheduler. The scheduling process is divided into two major parts: the *scheduling pipeline* and the *binding pipeline*, which are further subdivided into stages. Each stage provides an extension point for registering plugins. The scheduling pipeline consists of a sequence of filtering and scoring stages. Filter plugins remove nodes that are incapable of hosting a container, while score plugins assign a score to the nodes that have survived filtering. The node with the highest cumulative score is picked to host the container and admitted to the binding pipeline, which enacts this decision on the cluster.

The stages of the scheduling pipeline are depicted as white boxes in Figure 5.2. The `Sort` stage establishes the order in which the incoming containers will proceed through the scheduling pipeline – this stage supports only a single plugin. The `PreFilter` stage is executed once per container and is intended for caching information that needs to be computed once for the container and not for every candidate node. The `Filter` stage is executed for each candidate node and is responsible for removing nodes that are incapable of hosting the current container. `PostFilter` is only executed if no nodes are left after filtering – this stage allows, e.g., preempting other containers to then retry filtering. The `PreScore` and `Score` stages are the scoring counterparts to `PreFilter` and `Filter`. The `NormalizeScore` stage can be used to normalize a plugin's node scores to an integer between 0 and 100, which is required by the framework. Afterwards, the scores of all plugins are accumulated and the node with the highest score is selected. This selection is relayed to the `Reserve` stage, which allows plugins to update third-party data structures. `Permit`, the final stage of the scheduling pipeline, allows approving, denying, or delaying a container's entrance to the binding pipeline. Polaris Scheduler's scheduling pipeline focuses on providing Edge and SLO awareness. We assume that other scheduling requirements, such as requested resources, are addressed by the underlying base framework, e.g., the Kubernetes scheduling framework's default plugins [235].

5.3.2 Cluster Topology Graph and Service Graph

Unlike in a Cloud environment with a high-speed flat network structure, an Edge cluster node is often not "directly connected" to all other nodes, because an Edge cluster's

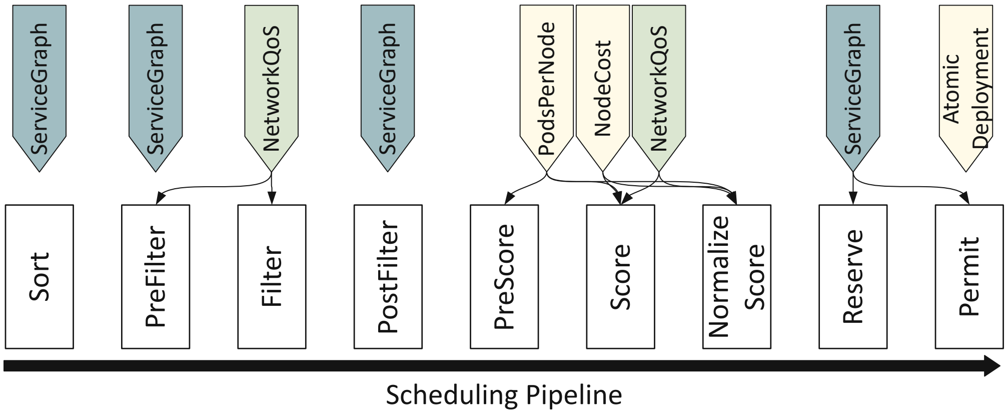


Figure 5.2: Scheduling Pipeline (based on [237]) and Polaris Scheduler Plugins.

network structure does not resemble a complete graph. Some nodes may be connected directly to each other, while other nodes might only be reachable through a gateway node. Furthermore, the types of network connections and their QoS properties may vary greatly; some connections are fast WiFi or 5G links, while others are slower, such as 3G. To capture the network topology of a cluster and the QoS properties of the connections, we use a *Cluster Topology Graph*, which is an undirected graph, where every node in the graph represents a node in the cluster and each link between two nodes represents the network connection between them. Each link is annotated with the QoS properties of this connection, i.e., its bandwidth, latency, bandwidth variance, latency variance (jitter), and the packet drop percentage. We have created a CRD to store information about each network link as an object in Kubernetes. We assume that these network link objects needed to build the Cluster Topology Graph are available to the scheduler and that they reflect a recent state of the network. The specifics of how these links can be generated and updated by monitoring solutions has no direct effect on Polaris Scheduler and is beyond the scope of this work.

To model the topology of an application and its network QoS requirements, Polaris Scheduler relies on a *Service Graph* [192], like the one in Figure 5.1. This is a directed acyclic graph (DAG), where each node represents a microservice of the application (all instances of this microservice are captured by a single node). A link from node α to β indicates that microservice α makes requests to microservice β. Each Service Graph link can be annotated with the minimum network QoS requirements for the network connection between the two microservices. Specifically, the Polaris Scheduler supports `minBandwidth`, `maxBandwidthVariance`, `maxLatency`, `maxLatencyVariance`, and `maxPacketDropBp`. All values are optional to allow developers to only configure those constraints that are important to their application. The Service Graph is implemented as a Kubernetes CRD, consisting of a list of node names and a list of link objects that use these node names and provide the previously mentioned network QoS configuration options. To denote its position in a Service Graph, a container can reference the Service Graph and the respective node by their names in its metadata. The scheduling framework and modeling support of Polaris Scheduler address RC-1 and RC-3, whereas the scheduler's plugins focus on RC-1 and RC-2. In the subsequent sections, we describe how these contributions are designed, implemented, and evaluated.

5.4 Polaris Scheduler Plugins

In this section we describe the main plugins of the Polaris Scheduler in detail. Figure 5.2 shows the Polaris Scheduler plugins as block arrows above the extension points of the stages of the scheduling pipeline. A block arrow with a green background indicates that the respective plugin provides optimizations that are based on highly dynamic data, like the Cluster Topology Graph, while a yellow background indicates that the plugin's optimizations are based on mostly static data, such as the hourly cost of a node. Finally, a turquoise background indicates that a plugin is of managerial nature and maintains shared data structures needed by the other plugins.

5.4.1 ServiceGraph Plugin

The `ServiceGraph` plugin is responsible for loading and maintaining the Service Graph to which a container is associated. It, thus, provides the foundation for almost all other plugins. To this end, it ties into multiple extension points.

Sort Stage. In this stage the plugin ensures that the containers that belong to the same Service Graph are scheduled in the order they are invoked upon a user request. This is necessary, because if network QoS constraints are specified for a link in the graph, Polaris Scheduler will ensure that a new container β, which is called by container α, is placed sufficiently close to container α to meet the QoS requirements. To this end, the scheduler needs to know where container α has been placed, hence the need for sorting. When the Sort stage is first invoked for a new container, for which the Service Graph has not been loaded yet, the `ServiceGraph` plugin fetches the Service Graph object from the cluster and places it in a shared cache within Polaris Scheduler. This cache uses reference counting to track how many containers are using a Service Graph. At this stage no scheduling context object has been created for the container yet, so each invocation of the `Sort` stage must look up the container's Service Graph in the local cache. The Kubernetes scheduling framework does not foresee any lengthy operations in the Sort stage, which may at some point lead to bad performance when handling a large number of containers. In our experiments (see Section 5.5) we did not notice any problems. Nevertheless, we will investigate the possibility of designing a custom scheduling pipeline as future work. This plugin stage also triggers the lookup of the cluster nodes with already running containers of this Service Graph, because the current container may not have been created during the initial deployment, but, e.g., as a result of horizontal scaling. This lookup is performed asynchronously and, thus, does not affect the performance of the `Sort` stage.

PreFilter Stage. At this stage the scheduling context object becomes available for the

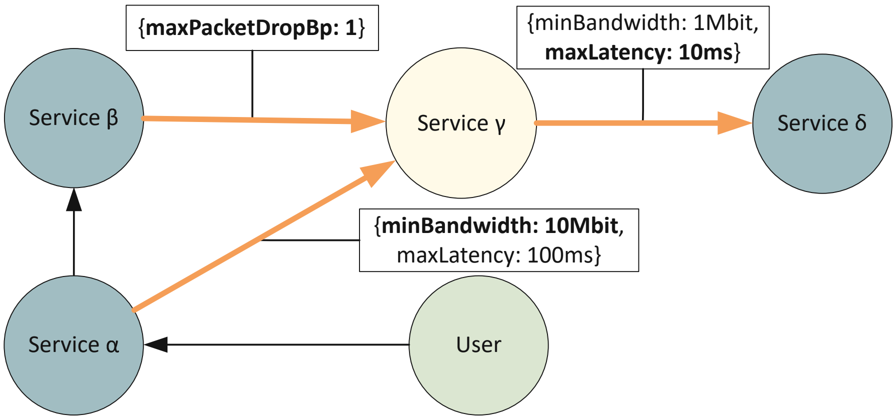

{maxPacketDropBp: 1}
{minBandwidth: 1Mbit, maxLatency: 10ms}
Service β
Service γ
Service δ
{minBandwidth: 10Mbit, maxLatency: 100ms}
Service α
User

Figure 5.3: Most Stringent QoS Requirements for Service γ.

container, so the plugin caches the container's Service Graph in the respective scheduling context.

Reserve Stage. This stage logs the placement of the container in the locally cached Service Graph, so that it can be looked up for containers that are still queued.

PostFiler, Unreserve, & Permit Stages. These stages decrement the reference count of the Service Graph in the shared cache and eventually release the graph object.

5.4.2 NetworkQoS Plugin

The `NetworkQoS` plugin filters out all cluster nodes that do not meet the network QoS requirements defined on the incoming and outgoing Service Graph links. The plugin supports throughput (bandwidth), latency, latency variance (jitter), and packet drop. Enforcing the requirements of the incoming Service Graph links entails looking up the cluster nodes of the already scheduled containers that represent the sources of these links. For each candidate node, the `NetworkQoS` plugin needs to calculate the shortest path between the source container and the candidate node. The plugin ties into four stages: `PreFilter`, `Filter`, `Score`, and `NormalizeScore`.

PreFilter Stage. This stage is run once for each container and consists of the two major steps described in Algorithm 5.1.

Step 1. Lines 3–9: The overall network QoS requirements for the container are computed, based on all incoming and outgoing links of its node in the Service Graph. This entails iterating through all these links and collecting the most stringent requirement for every configured network QoS property, as shown in Figure 5.3. This information is needed for the heuristic executed in step 1 of the Filter stage.

Step 2. Lines 10–19: The incoming Service Graph links are cached for the container's Service Graph node. For each such link, the set of cluster source nodes is computed. It

Algorithm 5.1 NetworkQoS PreFilter Stage

```
1:  Input: $G_S = (V_S, E_S)$: Service Graph;
        $\pi \in V_S$: Service Graph node corresponding to current container;
2:  Output: $R_\pi = (maxLatency_\pi, ...)$: Overall network QoS requirements for $\pi$;
        $E_{\chi,\pi}$: Incoming Service Graph links for $\pi$;
        $SRC$: Cluster nodes that host the source for each link in $E_{\chi,\pi}$;
        $P$: Shortest path trees for each $n \in SRC$;

3:  $R_\pi \leftarrow (maxLatency_\pi = \infty, ...)$                ▷ Init $R_\pi$ to most lenient values
4:  for all $e \in E_S$ involving $\pi$ do
5:      if $maxLatency_e < maxLatency_\pi$ then
6:          $maxLatency_\pi \leftarrow maxLatency_e$
7:      end if
8:      Proceed analogously for the other network QoS properties
9:  end for

10: $E_{\chi,\pi} \leftarrow$ All service links coming into $\pi$
11: $SRC \leftarrow \{\}$; $P \leftarrow \{\}$
12: for all $(\chi, \pi) \in E_{\chi,\pi}$ do
13:     $N \leftarrow$ All cluster nodes that host an instance of $\chi$
14:     $SRC \leftarrow SRC \cup N$
15:     for all $n \in N$ do
16:         $sp \leftarrow$ Compute latency-wise shortest path tree for $n$
17:         $P \leftarrow P \cup \{sp\}$
18:     end for
19: end for
```

consists of all cluster nodes that have a container, representing the source of the Service Graph link, scheduled on them, as shown in the left part of Figure 5.4. For each cluster node in this set, the shortest paths tree in terms of latency is computed.

Filter Stage. This stage is run once for every candidate node and consists of the two major steps described in Algorithm 5.2:

Step 1. Lines 4–7 discard the candidate node if its selection is likely to prevent downstream services from being scheduled: If none of the node's network links meets the overall network QoS requirements computed in `PreFilter` step 1, discard it. This heuristic considers the most stringent network requirements of all Service Graph links, incoming and outgoing. Applied to the network links of the candidate node (i.e., not to a path) it helps avoid situations like the following: In a Service Graph $\alpha \rightarrow \beta \rightarrow \gamma$, suppose service α has been scheduled. When scheduling service β, we find a cluster node that fulfills the requirements for $\alpha \rightarrow \beta$, but the target node's network connection is too slow for $\beta \rightarrow \gamma$. Since γ remains yet to be scheduled, we cannot check a concrete path, but we can at least ensure that the network connection of β's target node is potent enough, hence the need for this heuristic.

Step 2. Lines 8–17 ensure that a path to the candidate node meets the requirements for the current service: Iterate through all incoming Service Graph links that were cached

Algorithm 5.2 NetworkQoS Filter Stage

```
Input: $cn$: Candidate cluster node;
    $\pi$: Service Graph node node corresponding to current container;
    $R_\pi$: Overall network QoS requirements for $\pi$;
    $E_{\chi,\pi}$: Incoming Service Graph links for $\pi$;
    $SRC$: Cluster nodes that host the source for each link in $E_{\chi,\pi}$;
    $P$: Shortest path trees for each $n \in SRC$;
Output: $canHost$: $true$ if $cn$ can host $\pi$, otherwise $false$;
    $BW_{var}, L_{var}$: Max bandwidth & latency variances for shortest paths;

$canHost \leftarrow true$; $BW_{var} \leftarrow \{\}$; $L_{var} \leftarrow \{\}$
if $cn$ does not meet $R_\pi$ then
    $canHost \leftarrow false$
    return
end if

for all $e = (\chi, \pi) \in E_{\chi,\pi}$ do
    $sp \leftarrow$ FindShortestCompliantPath(e)
    if $sp \neq nil$ then
        $BW_{var} \leftarrow BW_{var} \cup \{$ highest bandwidth var in $sp\}$
        $L_{var} \leftarrow L_{var} \cup \{$ highest latency var in $sp\}$
    else
        $canHost \leftarrow false$
        return
    end if
end for

function FindShortestCompliantPath($e = (\chi, \pi)$)
    $shortestPath \leftarrow nil$
    $N \leftarrow$ Look up nodes that host $\chi$ in $SRC$
    for all $n \in N$ do
        $p \leftarrow$ Shortest path from $n$ to $cn$ from $P$
        if $p$ meets QoS requirements for $(\chi, \pi)$ then
            if $shortestPath = nil$ OR $p < shortestPath$ then
                $shortestPath \leftarrow p$
            end if
        end if
    end for
    return $shortestPath$
end function
```

in the `PreFilter` stage and for each link, examine the shortest path, latency-wise, from each cluster source node found in step 2 of the `PreFilter` stage to the candidate node. Pick the shortest path that meets all network QoS requirements. If none can be found, discard the candidate node. For example, in Figure 5.4 (left side) the service link $\alpha \rightarrow \gamma$ is examined. Service α is scheduled on the cluster nodes A and C. Node E is the current candidate node. The Cluster Topology Graph (right side of Figure 5.4) shows the shortest path from Node A to the candidate node (orange) and the shortest path from Node C to the candidate node (blue). The path from Node A fulfills the network QoS requirements, so it is picked. Finally, store the highest bandwidth and latency variance values of the picked path for the `Score` stage.

Score and NormalizeScore Stages. In the `Score` stage, the latency and bandwidth variance values of the picked paths are used to assess the stability of the network connections. A lower variance indicates a higher probability that network QoS will remain stable and, thus, results in a higher score (50% bandwidth variance, 50% latency variance). The `NormalizeScore` stage is used to clean up cached data.

5.4.3 Other Plugins

PodsPerNode Plugin

This plugin is inspired by one of the plugins of the Pogonip scheduler from Chapter 4. It assigns a score to cluster nodes, based on how many replicas of the container they would be able to host, depending on its configuration:

1. *More possible replicas* ⇒ *higher score* favors nodes with low resource utilization to avoid congestion by placing multiple containers in the same node.

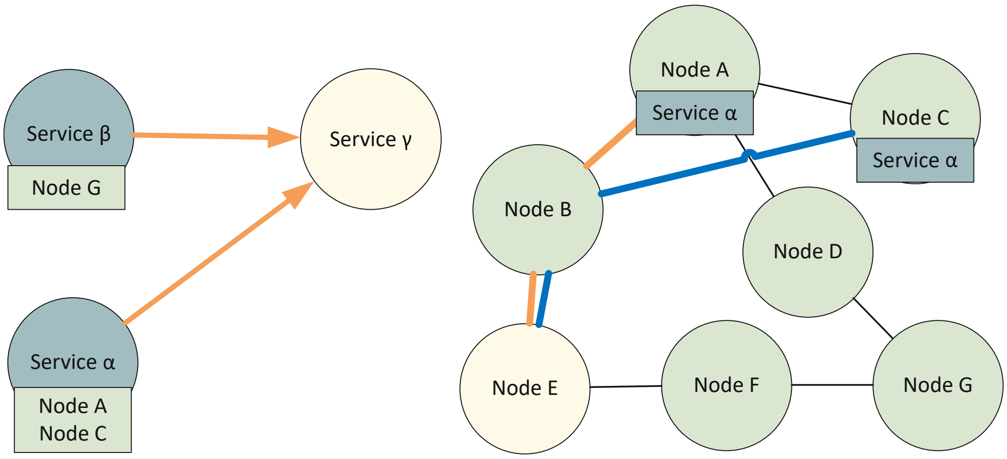


Figure 5.4: Incoming Service Links for Service γ (left), Shortest Network Paths to Candidate Node from Service α (right).

2. *More possible replicas* ⇒ *lower score* gives preference to nodes, which may already contain other components of the application and prefers using as many resources as possible on a smaller set of nodes instead of scattering containers across all nodes. This allows edge nodes, which are unused to go into a power conserving state.

NodeCost Plugin

This plugin assigns higher scores to cheaper nodes, which is often neglected by existing schedulers.

AtomicDeployment Plugin

This `Permit` plugin ensures that either all containers belonging to a Service Graph exit the scheduling pipeline successfully and enter the binding pipeline or, if at least one container fails to get a cluster node assigned, all other containers of this Service Graph will fail as well. This ensures that no resources are wasted on Edge nodes by containers that belong to an incompletely scheduled application. This plugin acts only upon the initial deployment of an application, but not on containers created due to scaling.

5.5 Evaluation

We evaluate Polaris Scheduler using the traffic analysis and hazard detection use case illustrated in Section 5.2. We describe our experiment setup in Section 5.5.1 and present the results in Section 5.5.2, followed by a discussion in Section 5.5.3.

5.5.1 Experiment Setup

For the experiments we specify the Service Graph shown in Figure 5.1, including the indicated network SLOs. Each microservice is represented by a Kubernetes Deployment object that defines the service's resource requirements (the used container images are irrelevant, since we benchmark the placement of the microservices and not the use case application itself):

- Collector: 1 vCPU, 1 GiB memory, and a 5G base station
- Aggregator: 4 vCPUs, 2 GiB memory
- Hazard Broadcaster: 2 vCPUs, 2 GiB memory
- Region Manager: 4 vCPUs, 8 GiB memory
- Traffic Info Provider: 2 vCPUs, 2 GiB memory

The Edge cluster is simulated using `kind`[5], a tool for running a Kubernetes cluster inside Docker, and `fake-kubelet`[6] for adding mocked nodes to this cluster. We run a single Kubernetes (v1.22.9) `kind` control plane node, which hosts core Kubernetes controllers and the schedulers. The nodes used as scheduling targets are simulated using `fake-kubelet` and are visible as ordinary Kubernetes nodes; their resources and other details are configurable via templates. A container assigned to such a node will enter the `Running` state, but it will not actually be executed, which is fine, because we benchmark the placement of the containers, not their execution. We run the experiments on a VM with 24 virtual CPU cores and 47 GiB of RAM. The hosting server has an Intel Xeon CPU (Cascade Lake) with a base clock of 2.1 GHz. Since the Kubernetes network proxy on each node reduces the CPU and memory quantities available for scheduling, even on `fake-kubelet` nodes, we rely on the extended resources mechanism of Kubernetes to set up `cpu` and `memory` resources, which are available for scheduling in their entirety.

We run two experiments: (i) a *Network QoS SLOs Compliance experiment* for assessing whether the container placement fulfills the network QoS SLOs of the application and (ii) a *Performance and Scalability experiment* for evaluating the schedulers when placing increasingly larger applications on growing cluster sizes. For the Network QoS SLOs Compliance experiment we deploy a small-scale version of the application consisting of three Collector instances and a single instance of each of the other services in a test cluster with 12 nodes. For the Performance and Scalability experiment, we deploy increasingly larger-scale versions of the application by multiplying the instance counts of all microservices, except for the Region Manager, with a multiplier $m = \{10, 20, \ldots, 70\}$. We do the same with the size of the Edge cluster. For example, for $m = 10$, we deploy 30 Collectors, a single Region Manager (which coordinates the other services), and 10 instances of each of the other microservices on a 120 nodes cluster.

We design the cluster for the Network QoS SLOs Compliance experiment and reuse the same topology to create m equal subclusters for the Performance and Scalability experiment. The topology of each subcluster and the nodes' resources are shown in the left part of Figure 5.5. Each subcluster consists of 11 Edge nodes, three of which have a 5G base station (indicated by the antenna icon), and a Cloud, which is modeled as a single large node with 16 CPU cores and 32 GiB memory. The resources of the Edge nodes resemble Raspberry Pi 3 Model B+ (4 CPU cores and 1 GiB of RAM) and Raspberry Pi 4 Model B (4 CPU cores and 4 GiB or 8 GiB of RAM)[7] devices and are named accordingly as `raspi-3b-ID`, `raspi-4s-ID` ("Raspberry Pi 4-small"), and `raspi-4m-ID` ("Raspberry Pi 4-medium"). All network links are annotated with their latencies and bandwidths. The links between `base-0` and `raspi-4m-3` and between `base-0` and `raspi-4s-0` are additionally marked with high bandwidth variance and low bandwidth variance respectively, indicating that the bandwidth of the former link is subject to great fluctuations, whereas the latter link does experience fluctuations, but

[5] `https://kind.sigs.k8s.io`

[6] `https://github.com/wzshiming/fake-kubelet`

[7] `https://www.raspberrypi.org`

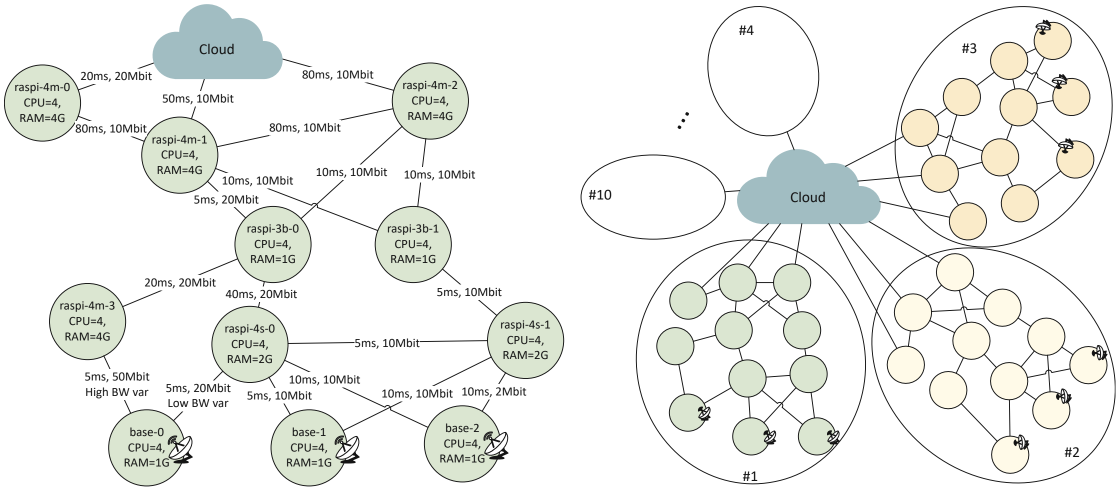


Figure 5.5: Edge Cluster Topologies: Network QoS SLOs Compliance Experiment (left); Performance and Scalability Experiment with 10 Subclusters (right).

they are much less. The bandwidth variances of the remaining links are negligible. To form the larger-scale test cluster, we replicate the nodes and links of a single subcluster m times and interconnect the subclusters through their Cloud nodes.

5.5.2 Experiment Results

We benchmark Polaris Scheduler against the default Kubernetes scheduler (kube-scheduler) and two theoretic approaches, Greedy First-fit and Round-robin. For each experiment configuration and scheduler we perform five iterations of deploying the application, recording the placement and the time required to place each container, and then undeploying the application again. The Greedy First-fit and Round-robin schedulers are implemented using the Kubernetes scheduling framework, by reusing all default `Filter` plugins to obtain the set of eligible nodes and then relying on a single `Score` plugin to assign the highest score to node chosen by the respective placement strategy. All scripts and configuration files needed to reproduce the experiments are available in our repository. All schedulers found placements for all microservices in both experiments. The default resource-related plugins of the Kubernetes scheduling framework ensured that only nodes that had the required resources were selected.

Network QoS SLOs Compliance

In this experiment, executed on the small-scale cluster in the left part of Figure 5.5, the main goal is to assess whether the placements computed by the schedulers fulfill the network QoS SLOs of the use case application. The link from the Collector to the Hazard Broadcaster is the most critical link, since the latter microservice broadcasts the existence of a hazard to nearby cars. Meeting the network SLOs of this service link (max latency of 10 ms and min bandwidth of 1 Mbps) is crucial for driver safety;

we will place special emphasis on whether this has been achieved by each placement. Figure 5.6 summarizes the average latencies between the microservices that were achieved by placements computed by the four schedulers across all iterations of this experiment. We use this average, because different iterations may yield different placements (not only for Round-robin) due to the reuse of many Kubernetes scheduling framework scoring plugins, which influence the placement. If multiple nodes have the same aggregated top score, a random one is picked from this set. The red line in the graph indicates the max latency SLOs, i.e., the upper bounds, for each service link. Figure 5.7 summarizes the average bandwidths between the microservices from the same experiment, with the red line indicating the min bandwidth SLOs, i.e., the lower bounds. For the links between the three Collectors and the single Hazard Broadcaster, as well as the single Aggregator, we compute the mean average across the three network paths to obtain the value for a single experiment iteration. If two interconnected microservices are placed on the same node, we consider them to have zero latency and a bandwidth of 100 Gbps.

The Greedy First-fit scheduler selects the first node that matches a container's resource requirements. The node iteration order is determined by the alphabetical sorting of the node names, such that it is the same across all runs. Thus, the Greedy First-fit scheduler computed the same placement on every iteration: one Collector was placed on each of the `base` nodes, while all other microservices were placed on the `cloud` node. This results in a total latency of 75 ms from the Collector to the Hazard Broadcaster, which is 7.5 times the upper SLO limit. The placement also violates the less stringent 50 ms max latency SLOs between the Collectors and the Aggregator – the lowest latency path of this link also violates the 10 Mbps min bandwidth requirement for one Collector instance, but this can be solved when taking an alternative network path (with even higher latency). The network QoS SLOs between the other microservices are met, since they all reside on the same node.

Round-robin operates on a circular list of nodes, based on the same iteration order as Greedy First-fit, and picks the first matching node encountered from the starting position. If a scheduling cycle ends at position n in the list, the next one will start from position $n + 1$. Round-robin used the same nodes on each of the five iterations: one Collector was placed on each of the `base` nodes, the Aggregator on the `cloud` node, the Hazard Broadcaster on `raspi-4m-0`, the Region Manager on the `cloud` node, and the Traffic Info Provider on `raspi-4m-1`. The reason for this is that the Collectors can only be assigned to the `base` nodes. Thus, after the Collectors have been scheduled, the next iteration will always point to the `cloud` node. The latency between the two safety critical microservices, Collectors and Hazard Broadcaster, is 95 ms, which is 9.5 times the SLO limit, thus, even worse than the one achieved by Greedy First-fit. Akin to Greedy First-fit, the max latency and the min bandwidth SLOs (on the lowest latency path) between the Collectors and Aggregator, as well as the min bandwidth SLO between the Region Manager and the Traffic Info Provider, are also not met.

The Kubernetes default scheduler, like Greedy First-fit, placed the Collectors on the `base` nodes and all other microservices on the `cloud` node. Thus, it also violates

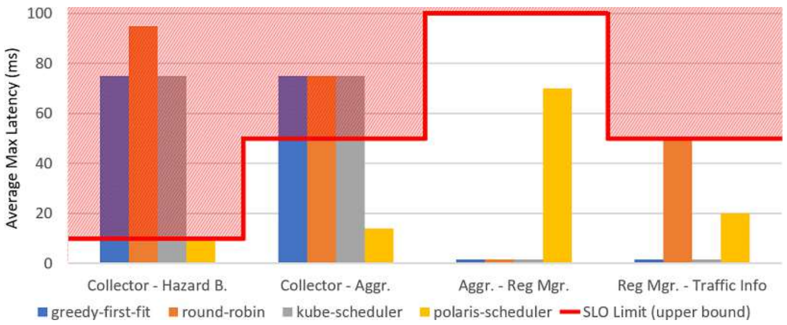


Figure 5.6: Average Max Latencies Achieved by Schedulers and SLO Bounds.

the safety critical max latency SLO between the Collectors and the Hazard Broadcaster (7.5 times the limit), as well as the max latency and min bandwidth SLOs between the Collectors and the Aggregator, albeit the min bandwidth SLO can be met by taking an alternative network path (with even higher latency).

Polaris Scheduler computed three different sets of placements. The Collectors were always assigned to the base nodes, the Region Manager to the cloud node, and the Traffic Info Provider to raspi-4m-0. The remaining two microservices were placed (i) once the Aggregator to raspi-4m-2 and the Hazard Broadcaster to raspi-4s-1, (ii) once the Aggregator to raspi-4m-2 and the Hazard Broadcaster raspi-4s-0, and (iii) three times the Aggregator to raspi-4s-0 and the Hazard Broadcaster to raspi-4s-1. All placements fulfilled the network SLOs. In many cases the total latencies between the Collectors and the Hazard Broadcaster, as well as between the Collectors and the Aggregator, remained significantly below the SLO limits. Since the specified network SLOs are treated as hard constraints, any violation would cause the scheduling of the particular container to fail, e.g., if no node can provide a sufficiently high bandwidth. When the Hazard Broadcaster was placed on raspi-4s-0, two of the Collectors had a total latency of only 5 ms (the SLO limit is 10 ms). The total latency between the Collectors and the Aggregator was either 25 ms (Aggregator on raspi-4m-2) or 10 ms (Aggregator on raspi-4s-0), much below the limit of 50 ms. We note that when the Aggregator was placed on raspi-4m-2, the path with the lowest latency (25 ms) from base-2 to raspi-4m-2 did not fulfill the bandwidth SLO of 10 Mbps from Collector to Aggregator. However, an alternative path with a latency of 30 ms, which was also within the max latency SLO limit of 50 ms, did meet the bandwidth requirement. Polaris Scheduler was the only scheduler, whose placements met all network SLOs. In some cases the other schedulers achieved better latencies and bandwidths, because they placed the respective services on the cloud node, which lead to violations of other SLOs, including the safety-critical max latency SLO between the Collectors and the Hazard Broadcaster, whereas Polaris Scheduler chose tradeoffs that fulfilled all SLOs.

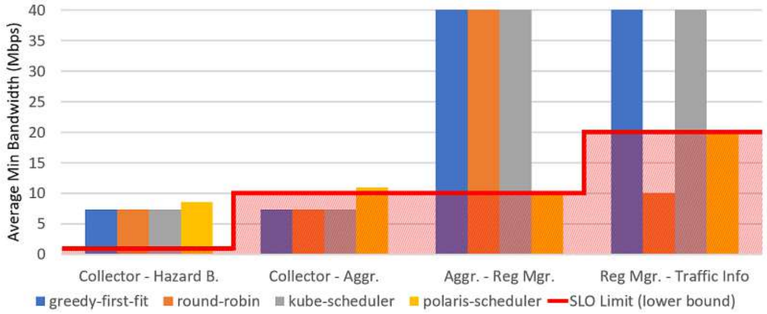


Figure 5.7: Average Min Bandwidth Achieved by Schedulers and SLO Bounds.

Performance and Scalability

Since a scheduler must be performant to ensure scalability, we now focus on the execution time required to place the entire application. We deploy our application in increasingly larger scales on clusters of increasing sizes, such as the one in the in the right part of Figure 5.5. We measure the execution times of the scheduling pipeline from the `PreFilter` stage until the `Permit` stage, for every container and compute the sum for all containers in an iteration. Waiting times, as introduced by the `AtomicDeployment` plugin are not included, because the aim is to reflect the computation time required for scheduling, not queuing time. We compare Polaris Scheduler to kube-scheduler. To have equal scoring conditions both schedulers are configured to score all nodes that passed filtering.

Figure 5.8 shows the scheduling times of both schedulers across application and cluster sizes. For scheduling 61 containers on 120 nodes Polaris Scheduler takes 346 ms, while kube-scheduler requires only 114 ms. For $m = 20$, i.e., 121 containers on 240 nodes, Polaris Scheduler requires $1,574$ ms, kube-scheduler needs 347 ms. This time difference can largely be attributed to the computation of the shortest path trees in the Cluster Topology Graph using Dijkstra's algorithm. Fetching the Service Graph for the first container of an application also consumes some time, but this becomes negligible as

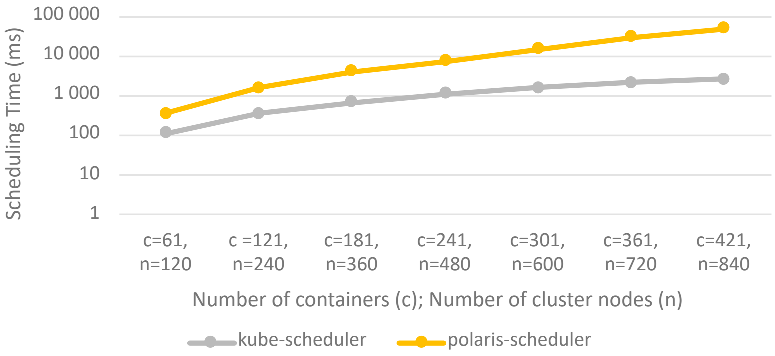


Figure 5.8: Scalability Analysis.

the application size grows. Despite the increased scheduling time due to the graph computations, Figure 5.8 shows that the scalability of Polaris Scheduler is comparable to that of kube-scheduler. Computing a placement with a focus on network SLOs comes at a cost, which is, however, acceptable for most long lived Edge applications, as we will discuss in the next Section.

5.5.3 Discussion

Schedulers must consider tradeoffs between multiple requirements. For Polaris Scheduler the most significant tradeoff is consciously accepting an increased scheduling time to allow finding a placement that fulfills the network SLOs. While very short lived applications (in the order of a few seconds) may not tolerate an increase in scheduling time with respect to kube-scheduler, Edge applications typically have a longer lifespan, e.g., the microservices of our use case run permanently. For an application that runs multiple hours or days, it is irrelevant if scheduling takes 100 ms or multiple seconds, if the placement fulfills the network SLOs. Such applications also normally do not arrive in large quantities, such that the scheduler would become a serious bottleneck. The second significant tradeoff in Polaris Scheduler concerns the Cluster Topology Graph. In very large clusters with thousands of nodes on a flat network structure, the graph could grow too big to store in memory or shortest path tree computations could take too long to be practicable. However, Edge clusters typically consist of many small subclusters that have a flat network structure within, but the subclusters themselves are sparsely interconnected, which makes using a Cluster Topology Graph feasible, considering the benefits that it yields. Nevertheless, we want to explore the use of a hypergraph as the Cluster Topology Graph in the future to drastically reduce the number of graph links to support large clusters with flat network structures. In such cases many nodes would pass the `Filter` stage, so scoring would need to be configured to operate only on a subset of these nodes, like in the default kube-scheduler configuration. Furthermore, we want to develop algorithms for distributed scheduling, to disperse the computational load required to schedule microservices on such large clusters.

In our Network QoS SLOs Compliance experiment only Polaris Scheduler fulfilled all SLOs. It is the only scheduler that considers the entire application and its SLOs, whereas the other schedulers ignored this information and placed each container independently of the others. While the other schedulers outperformed Polaris Scheduler on the latency between the Aggregator and the Region Manager, by placing both on the `cloud` node, they did so by violating the max latency SLO between the Collectors and the Aggregator. The minimum bandwidth SLOs were mostly met by the schedulers, even though a higher latency path was sometimes required. However, in an Edge environment, the link speeds may not be stable over time. The `NetworkQoS` plugin addresses the network dynamics found in an Edge cluster. By leveraging the Cluster Topology Graph, which needs to be maintained by an external monitoring service, it makes decisions not only based on the most recent measurements of latency, bandwidth, and packet drop, but also based on the variances computed from the recent bandwidth and latency history. These variances allow assessing how stable the node's connection has been over time, allowing Polaris Scheduler to compute a placement that not only fulfills the network SLOs at the moment, but that is likely fulfill them for a long time, thus, reducing the need for migrating a microservice to another node.

5.6 Summary

In this chapter we presented Polaris Scheduler, a network SLO-aware scheduler for Edge clusters. We motivated the need for SLO-aware scheduling using a realistic road traffic analysis and hazard detection use case. Polaris Scheduler models the complex dependencies and SLOs between the microservices of an application as a Service Graph, while the topology and current QoS state of the cluster is captured in a Cluster Topology Graph. Our scheduler leverages a multi-criteria decision making approach to find the most suitable compute node for a microservice.

The multi-criteria decision making is implemented as an extensible scheduling pipeline, where each criterion is provided by a plugin. The `ServiceGraph` establishes the order among the incoming microservices, based on their dependencies and provides information about the selected nodes for other parts of the application to the other plugins. The `NetworkQoS` plugin covers bandwidth, bandwidth variance, latency, latency variance, and packet drop. The consideration of bandwidth and latency variances allows selecting nodes that are likely to have stable network connections in the future. The remaining plugins optimize node resource usage, costs, and ensure that all microservices of the application can be deployed atomically.

By deploying our use case application on multiple Edge clusters, we evaluated Polaris Scheduler against kube-scheduler and two theoretical schedulers. We showed that the consideration of network SLOs during scheduling lays the groundwork for an application's fulfillment of its SLOs in heterogeneous Edge clusters.

CHAPTER 6

Vela: A 3-Phase Distributed Scheduler for the Edge-Cloud Continuum

The federation of multiple clusters in the Edge-Cloud continuum may quickly lead to tens of thousands of total compute nodes. Such large cluster sizes cannot be handled by monolithic schedulers and require a distributed approach. Vela Scheduler is a distributed scheduler that works in three phases. The first phase consists of an informed two-level sampling mechanism that delegates the sampling of target nodes to the potential target clusters to enable scalability. To ensure efficient sampling, these clusters leverage the job's requirements to return only nodes that are likely to be suitable as hosts. The second phase decides on the top three target nodes across all sampled clusters. The third and final phase commits the job to a target node according to the decision – to minimize rescheduling due to scheduling conflicts, committing is retried on the second and third best node if the previous one fails.

6.1 Introduction

Executing the microservices of an application on the right nodes of the Edge-Cloud continuum allows the application to take advantage of the best of both worlds, i.e., the low latency, proximity to the users, and attached IoT devices of the Edge and the powerful compute resources of the Cloud. Placing a workload in the Edge-Cloud continuum, which can often span tens to hundreds of thousands of nodes is challenging for a monolithic scheduler and, thus, often calls for a distributed scheduling approach.

This chapter is based on the paper T. Pusztai, S. Nastic, P. Raith, S. Dustdar, D. Vij, and Y. Xiong, "Vela: A 3-Phase Distributed Scheduler for the Edge-Cloud Continuum," in *2023 IEEE International Conference on Cloud Engineering (IC2E)*, 2023.

Table 6.1: Scheduler Architectures Comparison

Type	State per Instance	State Synchronization	Conflicts Handling	Limitations
Monolithic e.g., [246, 237, 56]	Entire cluster	Not needed, because single instance only	Avoided by monolithic state	Limited infrastructure size
Two-level e.g., [100, 244, 208]	Statically or dynamically partitioned by 1st level	Not needed, because state is partitioned	Avoided by partitioning	Local optima and potentially limited infrastructure size if 1st level is monolithic
Shared State e.g., [210, 22, 54, 77]	Entire cluster	E.g., read-only master state with frequent sync or partitioned sync	E.g., transactions or optimistic concurrency	Limited infrastructure size, since each scheduler maintains entire state
Distributed e.g., [173]	Sampled set of nodes	Sampling	Optimistic concurrency	Local optima
Hybrid e.g., [119, 53]	Depends on combination	Depends on combination	Depends on combination	One part is usually monolithic

There are multiple architectures for designing distributed schedulers, namely two-level, shared state, distributed, and hybrid [209]. We examine their differences from the monolithic architecture and from each other in four major aspects: i) how much of the scheduling-related infrastructure state a single scheduler instance sees, ii) how this state is synchronized, iii) how scheduling conflicts (i.e., two schedulers assign the same resources) are handled, and iv) architecture limitations.

Table 6.1 summarizes the scheduler architectures. Monolithic schedulers handle the entire infrastructure state within a single instance, which prevents conflicts, but limits scalability w.r.t. the infrastructure size. Two-level schedulers rely on a hierarchy, where the first level is responsible for the entire infrastructure state and statically or dynamically partitions it among an arbitrary number of second level schedulers. This prevents conflicts and improves scalability, but it may lead to local optima and, if the first level is monolithic, scalability may still be limited. Shared state schedulers operate with multiple schedulers that share access to the entire infrastructure state. Conflicts may occur, especially if the local state is outdated and the scale of the infrastructure is limited, because each scheduler has a copy of the entire state. Distributed schedulers rely on multiple schedulers that have a limited view of the infrastructure state, often obtained by selecting a portion of nodes (sampling), making this architecture highly scalable. The sampling algorithm influences the scheduling decisions' quality and the conflict probability. Hybrid schedulers combine two of the other approaches, usually a monolithic scheduler for one type of jobs and a distributed scheduler for all others.

Edge schedulers typically optimize placement for a set of Edge-specific constraints, such as network latency or geo-location, but they often lack the scalability needed for an online scheduler in the Edge-Cloud continuum, because they rely on computationally intensive algorithms, such as genetic algorithms, or because they focus on a single cluster and,

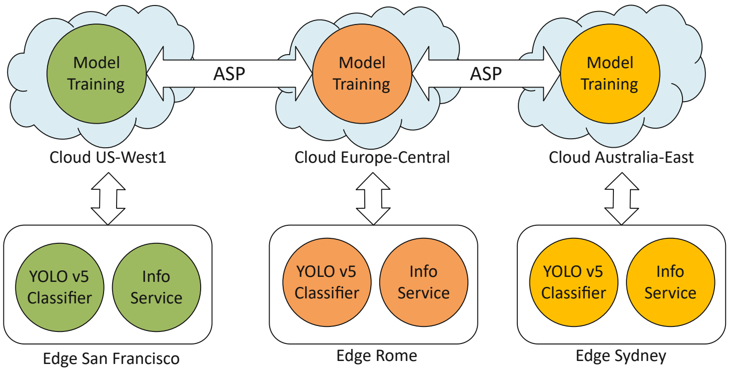


Figure 6.1: Globally Distributed Machine Learning.

hence, lack a distributed architecture. Those that focus on scalability, e.g., [95, 92, 181, 190, 188, 239], are often limited to scheduling batch jobs, not microservices, and none of them consider multiple globally distributed clusters. Their evaluations are limited to small clusters with less than 1,000 nodes, which does not allow drawing conclusions on global scalability.

Typically, clusters are managed by an orchestrator, e.g., Kubernetes[1] or Nomad[2], which is responsible for deploying and launching jobs and providing management services. The scheduler is often part of the orchestrator, but it may also be an external component that only interfaces with it to make job placement decisions.

The need for globally distributed scheduling in the Edge-Cloud continuum is exacerbated by novel large-scale applications that often require global deployments, such as general public AR/Metaverse [169] or geo-distributed ML. Such scenarios may also encompass scheduling on heterogeneous devices, like a combination of high-end servers and single-board computers, with the latter being required, e.g., for privacy preserving preprocessing [211].

A use case of globally distributed ML, based on the Gaia ML system [103], is shown in Figure 6.1. An AR application for tourists classifies images to display sightseeing information to its users. Classification jobs use the YOLOv5 CNN model to match user videos to sights in a city. An info service provides information to display to the users. Both jobs need to run as services in Edge clusters close to the users, because latency is critical in AR applications [189]. Training jobs to improve the model are run daily in a federated manner in the Cloud, relying mostly on local images from the closest Edge clusters and synchronizing the model globally using the Approximate Synchronous Parallel (ASP) model [103]. With global communications, latency plays a role and demands a reduction of packet round trips between scheduler and target nodes.

We formulate the following research challenges:

[1] `https://kubernetes.io`
[2] `https://www.nomadproject.io`

RC-1 *How can a scheduler for the Edge-Cloud continuum handle globally distributed Cloud and Edge clusters and scale reliably with the infrastructure?* As previously mentioned, monolithic schedulers can only handle a limited number of nodes, e.g., Kubernetes officially supports up to 5,000 nodes [236]. But also distributed schedulers may have limitations related to state synchronization, handling of scheduling conflicts, and scalability. However, scalability is an important feature of a scheduler [28], especially when dealing with very large infrastructures that span multiple, globally distributed clusters [46].

RC-2 *How can high-quality samples be collected from globally distributed clusters, while maintaining low sampling and scheduling latency?* Sampling-based schedulers are designed to handle large clusters. They commonly either retrieve samples from a local or shared cluster state, such as Tarcil [54], or contact nodes directly, like Sparrow [173]. The former approach does not work for globally distributed clusters, because maintaining a detailed state of globally distributed nodes is not feasible, nor does the latter, because contacting many globally distributed nodes directly would significantly increase scheduling latency, given global packet round trip times, e.g., 165 ms as per the Verizon SLA for a Europe-USA packet round trip [245] (sum of round trips within Europe, USA, and transatlantic). Additionally, as clusters get more loaded, it has been reported that larger samples are needed to find suitable nodes [54], because the samples are more likely to contain nodes that are full. Such wasted samples increase load on the scheduler. Thus, a sampling mechanism is needed that i) delegates work to the clusters to minimize the latency incurred by network communication and ii) leverages job requirements to return only suitable nodes to avoid an increase in sample size.

RC-3 *How can a distributed scheduler increase job throughput by reducing the number of scheduling conflicts?* The assignment of the same set of resources to two different jobs by two scheduler instances and the resulting conflict is an issue recognized by many distributed schedulers [210, 22, 54, 119, 77]. Rescheduling the conflicting jobs takes a significant amount of time and reduces the scheduler's job throughput, because the jobs need to traverse the entire scheduling lifecycle again. Reducing conflicts requires careful consideration of the scheduler's inner workings. While a job is being committed to a node, the sampling algorithm may rely on an outdated state and suggest a node, although it will be full after the commit has completed. Accounting for this issue and adding mitigation measures when conflicts do arise can significantly reduce rescheduling and, thus, increase the overall throughput of the scheduler.

In this chapter we present the open-source Vela Distributed Scheduler[3], which is part of Polaris SLO Cloud[4], a SIG of the Linux Foundation Centaurus project[5], a novel open-source platform for building unified and highly scalable public or private distributed

[3] https://polaris-slo-cloud.github.io/vela-scheduler

[4] https://polaris-slo-cloud.github.io

[5] https://www.centaurusinfra.io

Cloud and Edge systems. Vela continues our line of research on scheduling in the Edge-Cloud continuum continuum [165][186]. Our main contributions include:

1. *Vela Scheduler, a novel, globally distributed, orchestrator-independent scheduler with a 3-phase scheduling workflow* to enable optimized scheduling of microservices at global scale within the Edge-Cloud continuum. The workflow is distributed across multiple components to ensure scalability and is comprised of a sampling phase that retrieves node samples from globally distributed clusters, a decision phase that picks the best suitable node, and a commit phase that enforces the scheduling decision in a conflict-aware manner.

2. *2-Smart Sampling, a two-level, informed sampling mechanism that delegates sampling to globally distributed clusters and leverages job requirements to produce samples consisting of nodes that are likely to be suitable.* This reduces scheduling latency and sample wastage. Vela's design for globally distributed clusters delegates sampling to agents in the clusters, which frees the scheduler from communicating with the nodes directly. This delegation greatly reduces network traffic and latency for the scheduler. By leveraging job requirements, the likelihood that the sample contains suitable nodes is greatly increased, while avoiding large sample sizes, which would augment the scheduler's load. To the best of our knowledge, there is no other scheduler that is designed to perform sampling on a global scale or is evaluated in a globally distributed sampling scenario.

3. *MultiBind, a scheduling decision commit phase that automatically retries committing the job to another node if a scheduling conflict occurs*, without rerunning the entire scheduling process. This significantly reduces the number of jobs that need to be rescheduled due to conflicts and, thus, increases the overall throughput of the scheduler.

This chapter is structured as follows: Section 6.2 provides an overview of the architecture of the Vela Distributed Scheduler, and Section 6.3 describes the mechanisms that realize our contributions. Section 6.4 evaluates our scheduler on multiple interconnected Kubernetes clusters that represent an Edge-Cloud continuum and Section 6.5 provides a short summary of our work.

6.2 Vela 3-Phase Scheduling Workflow

The Vela Distributed Scheduler is designed to manage multiple, globally distributed Edge and Cloud clusters. It consists of two components, the *Scheduler* and the *Cluster Agent*. The scheduler can be deployed with an arbitrary number of instances, which are independent of the infrastructure, i.e., clusters, they need to manage. Due to its orchestrator-independent design, clusters may be operated by different orchestrators, e.g., one cluster might use Kubernetes, while another cluster might use Nomad. The

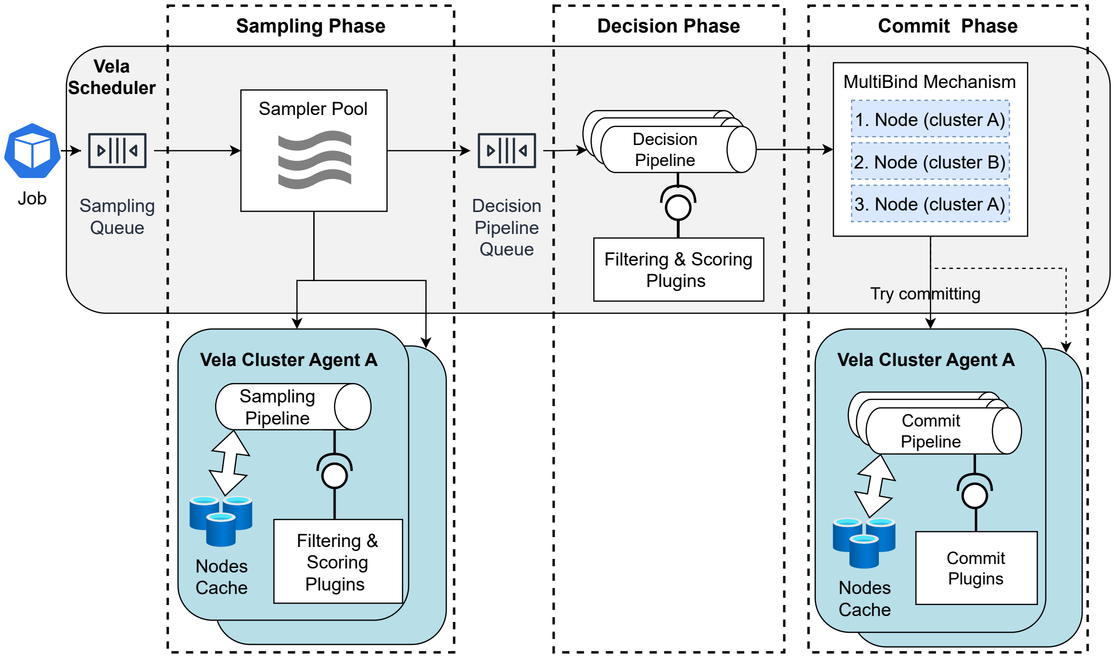


Figure 6.2: Scheduling Workflow and Job Lifecycle.

exact definition of a cluster node depends on the respective orchestrator – typically, a node will be either a VM, a bare-metal server, or a single-board computer. Every node can host multiple jobs, as long as it has sufficient resources to accommodate them. To become enabled for the Vela Scheduler, each cluster only needs to deploy the Cluster Agent, typically as a controller.

The 3-phase scheduling workflow (see RC-1) is the logical centerpiece of Vela. The workflow and the lifecycle of a job within it are shown in Figure 6.2. Each of the three phases, i.e., **sampling**, **decision**, and **commit**, contains a pipeline; these pipelines are shown in Figure 6.3. Each pipeline consists of multiple stages. The business logic within each stage is realized through plugins, which facilitates the implementation of diverse scheduling policies.

The 3-phase scheduling workflow starts when a user or another system component, such as an autoscaler, submits a job to an arbitrary instance of the Vela Scheduler. The scheduler instance sorts incoming jobs, e.g., based on priority, in its Sort stage and then adds them to its *Sampling Queue*.

Once the scheduler dequeues the job, it enters the *sampling phase* with the 2-Smart Sampling mechanism – the sampler pool can process multiple jobs in parallel in this phase. 2-Smart Sampling consists of two steps, the first one is executed by the Sample Nodes plugin. It selects a random subset of all configured clusters to be used for sampling and requests a sample from their respective Cluster Agents, passing the job's requirements along to ensure that only nodes that fulfill these requirements are returned.

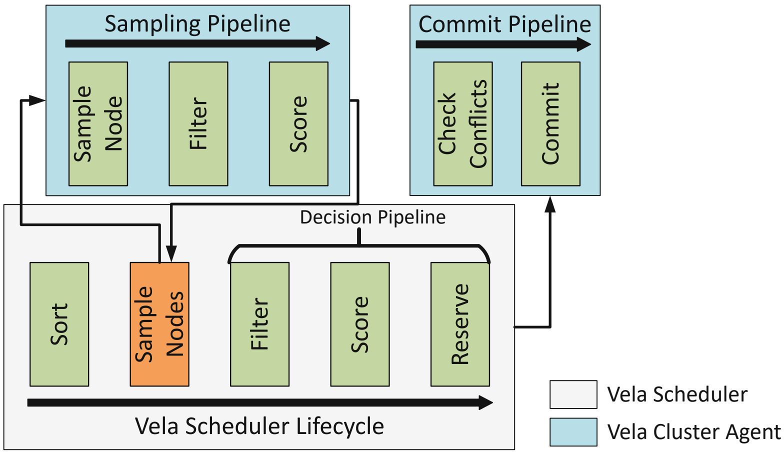


Figure 6.3: 3-Phase Scheduling Workflow with Sampling, Decision, and Commit Pipelines.

Each cluster's Cluster Agent, then, executes the second step of 2-Smart Sampling. The agent maintains a frequently updated cache of its cluster's nodes – the exact implementation depends on the underlying orchestrator, e.g., Kubernetes provides a watch mechanism that notifies the agent on nodes list changes. The agent selects a set of nodes from this cache and executes the *sampling pipeline*, which employs a MCDM approach, consisting of the Filter and the Score stages. *Filter* plugins remove nodes that are not suitable for hosting a job and *Score* plugins assign scores from 0 to 100 to the nodes that have survived filtering, based on how suitable they are. A higher score indicates better suitability for the job, e.g., empty nodes may score higher than partially loaded ones. The sampled nodes are then returned to the Vela Scheduler, which places them, together with the job, in the *decision pipeline queue*. The presence of this queue ensures that sampling, which may consume some time, can be executed on different threads from the decision pipeline. This allows avoiding situations where all threads might be blocked waiting for samples, while the CPU remains idle, even though it could be used for the decision pipeline. If desired, a timeout can be configured for each sampling request to a Cluster Agent – this can be used if a use case has stringent requirements on scheduling latency.

When the job exits this queue, it enters the decision phase. The *decision pipeline* further evaluates the sampled nodes for their suitability using another set of Filter and Score plugins that allow enforcement of global policies. Multiple decision pipelines, each responsible for a single job, are executed on concurrent threads. Since the Cluster Agent's sampling pipeline also includes scoring, each node already has a list of scores produced by the sampling Score plugins. The scores computed by the scheduler's Score plugins are added to this list. After all eligible nodes are scored, the decision pipeline accumulates the scores and picks the top m nodes with the highest score, with m being determined by the configuration of the MultiBind mechanism. The Reserve stage can be used by plugins to update internal data structures. At the end of the decision pipeline, the scheduler advances the top m nodes to the commit phase. The decision pipeline requires no synchronization with other pipeline- or scheduler instances, because the only point of synchronization is located in the subsequent commit phase and is handled by the Cluster Agent.

In the *commit phase* the MultiBind mechanism instructs the Cluster Agent, responsible for the cluster of the first of the m selected nodes, to commit the scheduling decision to the node. Since scheduling decisions can be made simultaneously by multiple scheduler instances, scheduling conflicts may occur, i.e., two jobs may be assigned to the same node by different scheduler instances, but the node only has enough remaining capacity to host one of them. To handle such conflicts we rely on an optimistic concurrency approach within the Cluster Agent, which checks for each job, if the resources are still available. In case of a conflict, the first job to arrive is committed to the node, the second job is rejected. To this end, the *commit pipeline* first reserves resources in the agent's cache to make them unavailable to the sampling pipeline, then fetches the current information about the node, checks if the constraints are still fulfilled, and, finally commits the decision by binding the job to the node. If the commit pipeline fails, the MultiBind mechanism takes the next best node from the list of m most suitable nodes and tries committing the job to that one. Only if all m nodes fail, will the job be considered as having a scheduling conflict, which requires rescheduling, i.e., running the entire scheduling workflow again. Our experiments show that the MultiBind mechanism reduces the number of conflicts by a factor of up to 10.

Currently, Vela Scheduler is aimed at stateless microservices. However, its plugin-based design allows adding plugins to support stateful microservices or batch jobs in the future.

Vela is fault-tolerant by design. The failure of a Cluster Agent means that its cluster is not available for scheduling, but does not affect the availability of other clusters. Since no coordination is needed among scheduler instances, the failure of one instance only requires users to submit new scheduling requests to another instance, but has no effect on the operational status of the overall system.

6.3 Vela's Main Scheduling Mechanisms

In this section we present the two most important scheduling mechanisms, i.e., 2-Smart Sampling and MultiBind in detail.

6.3.1 2-Smart Sampling

To reduce latency and avoid large sample sizes, even in loaded clusters (see RC-2), Vela Scheduler introduces 2-Smart Sampling, a two-step informed sampling approach, where the scheduler delegates sampling to the Cluster Agents in the selected clusters. This delegation frees the scheduler from communicating with globally distributed nodes directly, which would incur high latency, and allows sampling to take full advantage of the local information that is available within the cluster. Specifically, 2-Smart Sampling executes in two steps:

1. The scheduler picks a random subset of all configured clusters to be contacted for samples. Using only a subset ensures scalability and reduces scheduling conflicts. Then, the scheduler contacts the Cluster Agent of each selected cluster for a node sample, passing along all the job's requirements.
2. Each contacted Cluster Agent runs the sampling pipeline to pick a set of nodes and check them for eligibility for hosting the job. The nodes that are deemed eligible are scored and then returned to the scheduler.

The percentage of clusters to be sampled (C_p) and the percentage of nodes to sample per cluster (N_p) can be configured.

The sampling pipeline in the second step of 2-Smart Sampling consists of three stages (see Figure 6.3): Sampling Strategy, Filter, and Score. The scheduling policy of each stage is implemented by one or more plugins. Currently we provide two Sampling Strategy plugins (a sampling request specifies which one to use), one for random sampling and one for Round-Robin sampling, and three Filter plugins: `ReourcesFit` ensures that a node fulfills the job's resource requirements, `GeoLocation` allows a job to specify that it needs to run in a specific location, and `BatteryLevel` allows restricting a job to running on a node that has a minimum battery level (if the node has a battery) – the former two plugins also tie into the Score stage.

For each job 2-Smart Sampling operates as shown in Algorithm 6.1:

Step 1. Lines 3–9 execute the first step of 2-Smart Sampling, i.e., pick a random subset of all clusters and request a sample from their Cluster Agents. The returned samples are added to the decision pipeline queue, together with the job.

Step 2. Lines 10–25 execute the second step of 2-Smart Sampling in each involved Cluster Agent. Lines 13–20 constitute the sampling and filtering loop, which proceeds until enough eligible nodes have been found or a timeout is reached. Line 14 gets a set of nodes from the Sampling Strategy plugin, e.g., random sampling (default) or Round-Robin. Lines 15–19 run all Filter plugins on each sampled node to determine if it fulfills the job' requirements. Lines 21–23 execute all Score plugins on each eligible node. Subsequently, the complete cluster sample is returned to the scheduler.

This approach ensures that each cluster's sample only contains nodes that meet the job's requirements, which allows for a smaller sample size. The sampling pipeline plugins need to ensure that the job's resource requirements are met by a node, but they may also implement complex policies that further improve the quality of the sample. The Cluster Agent may also implement cluster-specific scheduling policies.

Algorithm 6.1 Sampling Phase

```
Input: $j$: The job for which to sample nodes;
    $C_p$: Percentage of clusters to sample;
    $N_p$: The number of nodes to sample per cluster;
    $strat$: The sampling strategy to use;
Output: $S_e$: The set of sampled nodes that are eligible for hosting $j$ and their scores;

                          ▷ The 1st step of 2-Smart Sampling runs within the scheduler
$S_e \leftarrow \{\}$
$C \leftarrow$ PickRandomClustersToSample($C_p$)
for all $c \in C$ do
    $S_{e,c} \leftarrow$ c.RunClusterSamplingPipeline($j, N_p, strat$)
    $S_e \leftarrow S_e \cup S_{e,c}$
end for
AddToDecisionPipelineQueue($j, S_e$)

        ▷ The 2nd step of 2-Smart Sampling, i.e., the sampling pipeline, runs within a Cluster Agent
function RunClusterSamplingPipeline($j, N_p, strat$)
    $sampleSize \leftarrow$ ComputeSampleSize($N_p$)
    $S_{e,c} \leftarrow \{\}$                           ▷ The sampled nodes from this cluster

    while $|S_{e,c}| < sampleSize$ AND NOT timeout occurred do
        $S_i \leftarrow$ SampleNodesWithStrategy($strat$, $sampleSize$)
        for all $n \in S_i$ do
            if RunAllFilterPlugins($n$) $= true$ then
                                  ▷ If the node survives all filter plugins, it is eligible.
                $S_{e,c} \leftarrow S_{e,c} \cup \{n\}$
            end if
        end for
    end while

    for all $n \in S_{e,c}$ do
        RunAllScorePlugins($n$)      ▷ Run all score plugins and add the scores to the node $n$
    end for

    return $S_{e,c}$
end function
```

6.3.2 MultiBind Commit Phase

Vela Scheduler relies on an optimistic concurrency approach to deal with multiple decision pipeline or scheduler instances running in parallel. No cluster node resources are locked during the sampling phase, because most of them will not be used – in the end the job is assigned to a single node. This improves scalability, but entails that when committing a scheduling decision, another decision pipeline or scheduler instance may have already claimed some of the required resources on the node, resulting in a scheduling conflict for the current job. This is a common issue in distributed scheduling, which is normally handled by rescheduling the job (see RC-3) [210, 22, 54, 77]. In Vela Scheduler we mitigate this issue by the randomness in both steps of 2-Smart Sampling. Nevertheless, scheduling conflicts can occur. Note that the number of jobs per node is not limited, i.e., if the selected node has enough resources for both jobs, both are committed and executed – a conflict only occurs, if the node does not have sufficient resources for hosting both jobs.

To further reduce the number of scheduling conflicts that require rescheduling, Vela Scheduler relies on its MultiBind commit phase: instead of trying to commit the job only to the highest scored node and rescheduling it, if a conflict occurs, we use a list of the m highest scored nodes and try committing to the next node. Only if committing to all m nodes fails, the job is considered to have a scheduling conflict and needs to be rescheduled. Our tests in Section 6.4 show that a setting of $m = 3$ reduces the scheduling conflicts by factor of 10 compared to not using MultiBind. When trying to commit a job to a node, the MultiBind mechanism contacts the Cluster Agent of the node's cluster to execute the commit pipeline. This pipeline, which supports running multiple instances in parallel, contains two stages, whose logic is implemented using plugins: the *Check Conflicts* stage and the *Commit* stage. The entire process executed by the MultiBind commit phase is shown in Algorithm 6.2:

Step 1. Lines 3–8 represent the MultiBind mechanism, which executes in the scheduler. It iterates over the list of the m highest scored nodes and tries to commit the job to every node, stopping and reporting a scheduling success if the commit succeeds. If all commits fail, a scheduling conflict is reported. Each commit attempt, triggers the commit pipeline in the respective Cluster Agent.

Step 2. Line 10 proactively reserves the job's resources in the nodes cache to make them unavailable for sampling requests. Free resources that are not required by the job are still available for sampling.

Step 3. Line 11 locks the target node within the Cluster Agent such that no other commit pipeline can access it. Unreserved resources on the node are still available for sampling.

Step 4. Lines 12–14 fetch the target node and all jobs currently assigned to it from the orchestrator and, together with information from the nodes cache, compute the currently available resources on the node.

Algorithm 6.2 Commit Phase

```
 1: Input: $j$: The job to commit;
    $N = (n_1, ..., n_m)$: The $m$ highest scored nodes as commit candidates;
 2: Output: $SUCCESS$ or $CONFLICT$;

                                  ▷ The MultiBind mechanism runs within the scheduler
 3: for all $n \in N$ do
 4:     if RunClusterCommitPipeline($n, j$) = $SUCCESS$ then
 5:         return $SUCCESS$
 6:     end if
 7: end for
 8: return $CONFLICT$                      ▷ There was a conflict for all nodes in $N$.

                                  ▷ The commit pipeline runs within the Cluster Agent
 9: function RunClusterCommitPipeline($n, j$)
10:     ReserveResourcesInCache($n, j$)
11:     Lock($n$)

12:     $n^* \leftarrow$ FetchNodeInfo($n$)
13:     $J \leftarrow$ FetchJobsOnNode($n^*$)
14:     $n^* \leftarrow$ ComputeAvailableResources($n^*, J$)

15:     if RunCheckConflictsPlugins($j, n^*$) = $CONFLICT$ then
16:         UnreserveResourcesInCache($n, j$)
17:         $result \leftarrow CONFLICT$
18:     else
19:         Commit($j, n^*$)
20:         $result \leftarrow SUCCESS$
21:     end if

22:     Unock($n$)
23:     return $result$
24: end function
```

Step 5. Lines 15–17 execute the Check Conflicts plugins to check for a scheduling conflict. If there is a conflict, we undo the resources reservation in the nodes cache carried out in step 2 and prepare to report the conflict to the scheduler.

Step 6. Lines 19–20 run the Commit plugin to commit the job to the node.

Step 7. Lines 22–20 unlock the target node in the Cluster Agent to make it available to other commit pipeline instances again and then return the result to the scheduler.

Reserving resources in the nodes cache is a critical step, because otherwise the sampling pipeline would consider them still available, even though they are currently being bound to a job. Fetching the target node and its assigned jobs is needed, because the nodes cache could be outdated. The Commit stage first creates the orchestrator-specific job object and then binds it to the target node, which completes the commit pipeline.

6.4 Evaluation & Implementation

To evaluate our scheduler we focus mainly on the scalability aspect at a global scale, while keeping low latency and reducing scheduling conflicts, as described in our contributions. All code to run the experiments, as well as, all results can be found in our repository[6].

6.4.1 Implementation

Vela Scheduler and its Cluster Agent are implemented in Go; all their APIs are JSON-based REST APIs. The two largest engineering challenges lie within the Cluster Agent. The first one is the nodes cache, which needs to support a very high read frequency from sampling, but also a considerable write frequency stemming from the commit pipeline and orchestrator updates. The cache supports read-write locking, but to avoid holding locks for a long time, we treat all node objects as immutable. Reading is only done at three points: at the beginning of the sampling pipeline, by the Sampling Strategy plugins, and at the beginning of the commit pipeline. Writing is also done at three points: once at the beginning and at the end of the commit pipeline and when there is a node status update from the orchestrator. The second major engineering challenge is to reserve resources in the nodes cache as early as possible in the commit phase. It is critical to do this immediately for all incoming jobs, before locking the node (this locking only applies to the commit pipelines, not the nodes cache), because otherwise sampling would still consider resources as available, which will be consumed by a job waiting to be committed. At the end of the commit pipeline, each resource reservation is either committed or removed, depending on the outcome of the pipeline. Further implementation details can be found in our code repository.

6.4.2 Experiments Setup

To evaluate the scalability of Vela Scheduler we set up 10 globally distributed Kubernetes clusters, which vary in size, depending on the experiment. We run a single instance of Vela Scheduler, which, however, does not limit the distributed nature of our scheduler, because i) the 2-Smart Sampling mechanism is fully distributed and ii) each scheduler instance runs multiple sampling, decision, and commit pipelines independently of each other in parallel, which is the same as running multiple scheduler instances.

To set up the clusters in our testbed we use 10 Google Cloud Platform (GCP) VMs of type `c2-standard-8`, each having 8 vCPUs and 32 GB of memory and running on a physical machine with an Intel Cascade Lake or later processor. Every VM is bootstrapped with Ubuntu 22.04, on top of which we install MicroK8s[7] v1.25.6 to initialize a distinct single-node Kubernetes cluster. For all experiments, we rely on `fake-kubelet`[8] to create simulated nodes in each MicroK8s cluster. The resource properties of these nodes can

[6] https://polaris-slo-cloud.github.io/vela-scheduler/experiments

[7] https://microk8s.io

[8] https://github.com/wzshiming/fake-kubelet

Table 6.2: Node Types in Cloud and Edge Clusters.

	Node Type & Occurrence (%)	**vCPUs**	**RAM**	**Regions**
Cloud	50% cloud-small	2	4	Belgium, Oregon, Finland
	30% cloud-medium	4	8	
	20% cloud-large	8	16	
Edge	20% Raspberry Pi 4B	4	2	Belgium, Netherlands, Frankfurt, Montreal, Oregon, Finland, Iowa
	40% Raspberry Pi 4B	4	4	
	30% Raspberry Pi 3B+	4	1	
	10% cloudlet	4	8	

be easily configured and they are treated as ordinary nodes by Kubernetes. However, `fake-kubelet` nodes do not actually execute any pods (i.e., jobs), but this is not needed for our experiments, since we benchmark the scheduling performance, i.e., until a job has been bound to a node. Sampling performance is also not affected by `fake-kubelet`, because our sampling algorithm works against the Cluster Agent's nodes cache, which is maintained in the background. Other than consuming some CPU time on each VM, the use of `fake-kubelet` does not impact the metrics evaluated in these experiments.

Since Vela Scheduler focuses on the Edge-Cloud continuum, the 10 clusters are intentionally not homogeneous. We simulate three Cloud and seven Edge clusters with different types of nodes; the hosting VMs are located in different regions. Cloud clusters are made up of a combination of VMs of three different sizes and Edge clusters consist of a combination of Raspberry Pi[9] single-board computers and cloudlet servers. The node details, the percentage of each node type in the composition of a cluster, and the cluster regions are listed in Table 6.2. These node details serve as realistic configurations for the resource properties of the simulated nodes. There is no difference between simulating a cloud node and a Raspberry Pi node using `fake-kubelet`, because for our experiments only the configured resource properties are of interest. Vela Scheduler itself is also deployed on a `c2-standard-8` VM and is located in the Zurich region. All tests use Apache JMeter[10] as a load generator – we run it on a VM with 24 vCPUs and 47 GiB of RAM. The hosting server at our university has an Intel Xeon CPU (Cascade Lake) with a base clock of 2.1 GHz. JMeter does not allow for configuring a specific request rate per second, but instead requires configuring the number threads that generate requests and the approximate timing they should use, e.g., one request every 10 milliseconds.

We run three sets of experiments: i) *configuration tuning* to find optimal settings for Vela Scheduler, ii) *scalability with respect to infrastructure* to assess the performance of our scheduler on an increasing number of nodes, and iii) *scalability with respect to workload* to a assess the performance on an increasing scheduler workload.

Configuration Tuning aims to find optimal values for C_p and N_p, i.e, the percentage of

[9] `https://www.raspberrypi.org`
[10] `https://jmeter.apache.org`

clusters and the percentage of nodes to sample in 2-Smart Sampling. To this end, we evaluate settings of $C_p = \{10\%, 20\%, ..., 100\%\}$ and, for each value, run an experiment iteration with $N_p = \{4\%, 8\%, 12\%, 16\%\}$, for a total of 40 iterations. Each iteration tries to schedule $11,200$ jobs requiring 4 vCPUs and 4 GiB of RAM on clusters of $2,000$ (2K) nodes each, adding up to 20K nodes in total. $11,200$ is the maximum number of jobs of this size that this cluster configuration can support, thus the scheduler must find all available space to avoid scheduling failures. Additionally, 50% of the nodes are too small to host such a job.

The two scalability experiments use the settings determined by the configuration tuning to evaluate the scalability of Vela Scheduler. Akin to the previous experiment, each scalability experiments uses 10 clusters, each of which contains a tenth of the total nodes in the experiment, i.e., for 1K total nodes each cluster contains 100 nodes and for 20K total nodes each cluster contains 2K nodes.

The experiment on scalability with respect to infrastructure schedules 1K jobs on increasing cluster sizes, specifically 1K, 5K, 10K, 15K, and 20K total nodes (for comparison, Kubernetes officially only supports 5K total nodes [236]). We run three iterations for each of these cluster sizes. The jobs intentionally fit on each node to allow us to focus on measuring the execution performance.

The scalability experiment with respect to workload operates on 20K total nodes (i.e., 2K nodes per cluster) and gradually increases the scheduler workload across 15 iterations, each lasting three minutes. In this experiment the jobs are heterogeneous; specifically each JMeter thread iteration creates one job requiring 1 CPU and 1 GiB, two jobs needing 2 CPUs and 2 GiB, and one job requiring 4 CPUs and 4 GiB of RAM. We intentionally use CPU and RAM requirements only, because adding battery or geo-location requirements would reduce the number of eligible nodes and, hence, saturate the clusters sooner. Each job counts as one scheduling request. We use thread and timing configurations for JMeter to achieve job rates between 15.18 requests/sec and 290.36 requests/sec.

6.4.3 Results

Configuration Tuning

For this experiment we focus on finding the lowest values for C_p and N_p that yield zero scheduling failures. We aim for the lowest configuration values, because sampling fewer (globally distributed) clusters and fewer nodes within each cluster naturally leads to faster execution times than sampling more clusters and/or nodes. Since rescheduling attempts are common in distributed schedulers, we consider a job to have failed scheduling, only after having attempted rescheduling a total of ten times without success.

Figure 6.4 shows the number of scheduling failures as a percentage of the total jobs. It is evident that the number of failures decreases as the number of sampled clusters increases, because the scheduler has more nodes to choose from. The failures first reach zero at $C_p = 50\%$ and $N_p = 4\%$, which is what we will use for the remaining experiments. At

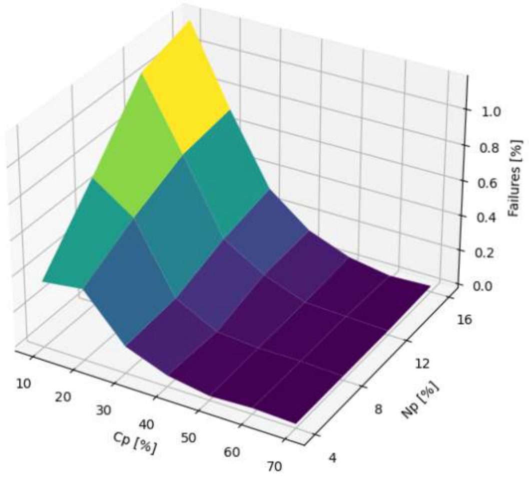


Figure 6.4: Scheduling Failure Percentages for Configuration Tuning.

$C_p = 60\%$ and $N_p = 4\%$, there is a single failure, but starting at $C_p = 70\%$, there are no more failures, which is why we have excluded larger C_p values from Figure 6.4 for clarity. The full set of results, including the number of rescheduling attempts, is available in our repository. For the remainder of this chapter we use the above mentioned lowest C_p and N_p values that resulted in no failures, i.e., perfect scheduling, within this experiment. However, future work may investigate dynamic adaptation of these values, because as the utilization of the clusters increases or decreases, different C_p and N_p values may be needed to maintain a low number of failures and scheduling conflicts.

Scalability with Respect to Infrastructure

This experiment focuses on evaluating the performance of Vela Scheduler on increasing cluster sizes to show its scalability. We examine execution times of the sampling phase, the commit phase, and the end-to-end (E2E) times, i.e., the time from adding a job to the sampling queue until a successful end of the commit phase. Since we noticed a significant latency increase of the MicroK8s API server under high load (e.g., creating a pod object sometimes took about 8 seconds), we do not include the interaction with Kubernetes in the commit and E2E metrics, instead we fetch node information only from our cache and consider the commit pipeline successful once we make the commit in our cache, before we issue a write request to the orchestrator. This allows us to focus solely on the Vela Scheduler performance.

Figure 6.5 summarizes the mean execution times in this experiment, showing a linear increase for all of them. We observe two different E2E times: one including time spent in the sampling queue (E2E) and one without sampling queue time (E2E-no-queue or E2E-nq). When including queuing time, E2E time increases much faster, albeit still linearly. This is because as the sampling duration increases, the threads responsible for step one of 2-Smart Sampling in the scheduler are blocked for a longer time. Since we

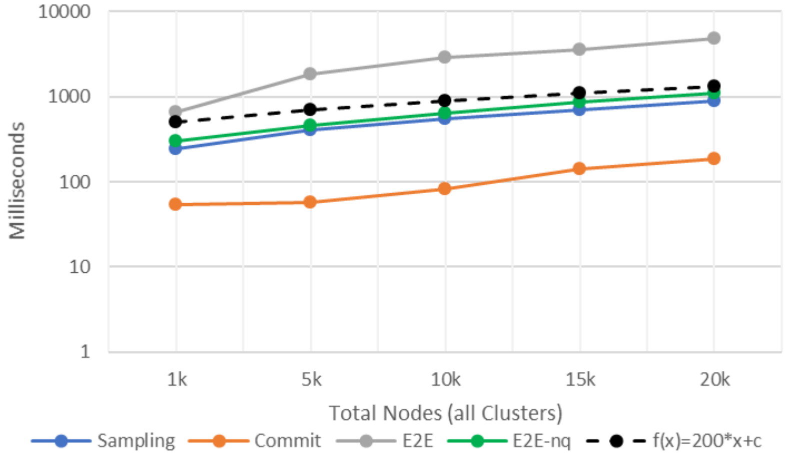


Figure 6.5: Mean Scheduling Times (ms) at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

have 80 sampling threads (CPU cores × 10) in the experiment, these threads are at some point all waiting for responses and thus many of the $1,000$ jobs that arrive in very quick succession need to stay in the queue longer. This could be alleviated, e.g., by running multiple concurrent scheduler instances.

More detailed breakdowns of the sampling, commit, and E2E-nq times are shown in Figure 6.6, Figure 6.7 and Figure 6.8 respectively. For 1K total nodes, sampling takes a mean of 243.3 ms, which is a reasonable time for getting samples from five globally distributed clusters, considering global packet round trip times (e.g., the Verizon SLA for a Europe-USA packet round trip, including intra-Europe and intra-US round trips is 165 ms [245]). Sampling times increase linearly with the cluster sizes to a mean of 902.1 ms for 20K total nodes. Since the Cluster Agent performs sampling on its nodes cache, which is regularly updated in the background, no communication within the cluster

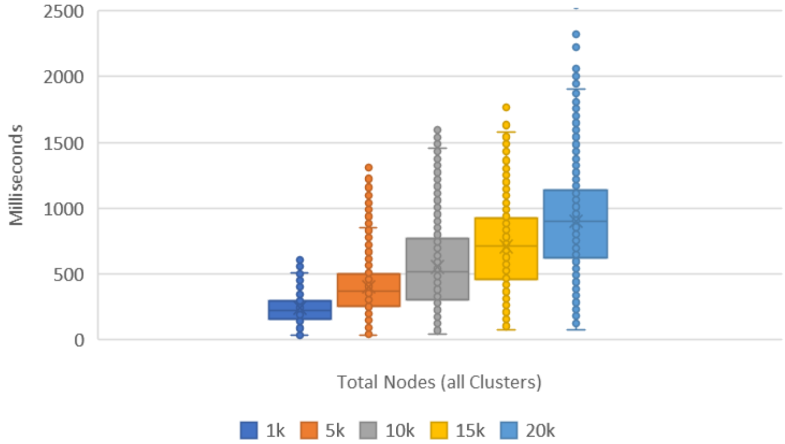


Figure 6.6: Sampling Times (ms) at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

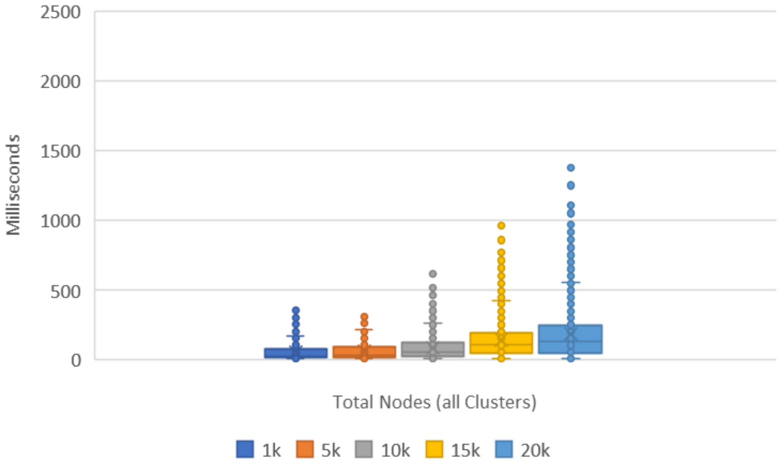


Figure 6.7: Commit Times (ms) at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

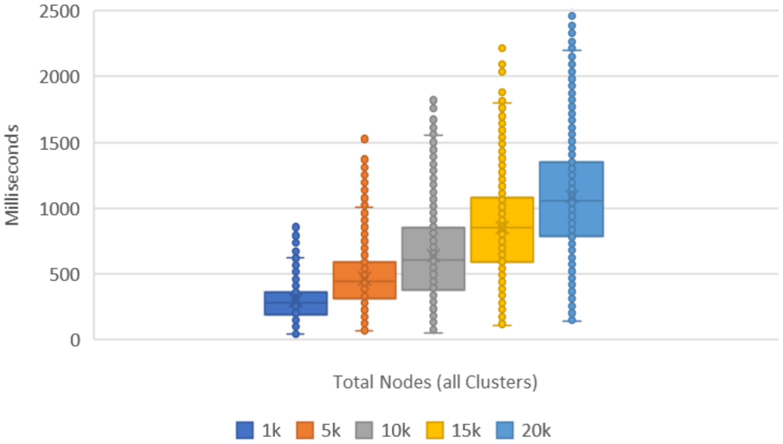


Figure 6.8: End-to-End Times (ms), without Sampling Queue, at $C_p = 50\%$ and $N_p = 4\%$ for Total Nodes.

is necessary in this phase. However, as the cluster size increases, the absolute number of nodes per sample also increases, hence more processing time is needed for larger clusters. Commit times increase linearly as well, ranging from 53.1 ms for 1K nodes to 182.8 ms for 20K nodes. Since the commit phase involves only communication with the target cluster, conflicts checking for a single node, cache operations, and possible MultiBind retries, its contribution to the E2E time is fairly low. E2E-nq times also increase linearly from 297.9 ms for 1K nodes to 1087.1 ms for 20K nodes. This shows that most of the time is spent in 2-Smart Sampling, which is reasonable given that all Filter and Score plugins currently run as part of the sampling pipeline.

The MultiBind overhead when trying to commit to all $m = 3$ nodes, compared to succeeding on the first node, varies depending on the communication latency with the selected clusters. However, it is evident from the execution time results that MultiBind

Table 6.3: Scheduling Decisions and Throughput.

Req / sec	Queuing Time (msec)	Scheduling Decisions/sec (SDPS)	Throughput w MultiBind (jobs/s)	Throughput no MultiBind (jobs/s)
54	0	54	54	49
72	1	72	72	62
94	6	95	94	75
99	106	100	98	73
133	30,097	110	107	77
175	35,499	238	96	87
212	35,672	384	99	94
254	32,562	608	116	112
290	30,847	817	134	131

provides considerable time savings over the alternative strategy of rerunning the entire Vela Scheduler lifecycle on every scheduling conflict, because this would encompass not only contacting at least one more cluster for committing, but also running the complete sampling phase again.

Scalability with Respect to Workload

In this experiment we evaluate all results with focus on the scheduler's *throughput* in *jobs per second (jobs/s)* and the total number of *scheduling decisions per second (SDPS)*. We calculate the throughput by dividing the number of successfully scheduled jobs by the total time the Vela Scheduler was active. This time is calculated using the difference between the scheduling finish timestamps of the last successful job and the first successful job. We compute this value for every iteration of our experiment and round it to the next integer value, giving us a throughput ranging from 15 jobs/s up to 134 jobs/s. The scheduling decisions per second (SDPS) are the total number number of scheduling attempts irrespective of their results (i.e., success, conflict, rescheduling due to no nodes found, or failure due to too many rescheduling attempts) divided by the total execution time. The SDPS range from 15 to 817. We stopped our experiments at this number, because the simulated cluster resources were getting exhausted, thus, leaving too little space for scheduling other jobs.

Table 6.3 summarizes the results of this experiment. It shows the request rate generated by JMeter in requests per second (req/s), the mean queuing time of a job before it is dequeued by the sampling pipeline, the SDPS, and the throughput in successfully scheduled jobs per second with and without MultiBind. The mean queuing time and the SDPS are good indicators of whether the scheduler is able to keep up with the incoming workload. Up until 99 req/s the queuing time is negligible, whereas starting with 133 req/s it suddenly rises to 30 seconds. Likewise, the SDPS are equal to or greater than the request rate up until 99 req/s and start lagging behind at 133 req/s. The

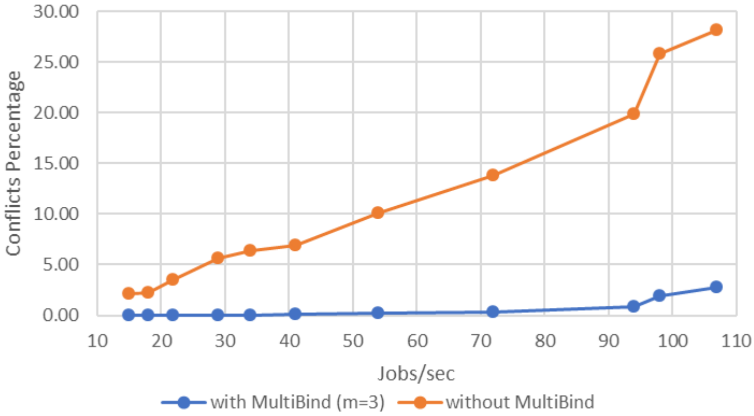


Figure 6.9: Scheduling Conflicts with and without MultiBind.

throughput with MultiBind remains approximately equal to the input request rate (the difference of 1 in the row with 99 req/s is caused by rounding, the actual difference is less than 0.042), until 133 req/s, where it starts to fall behind. These values indicate that the single-instance configuration of Vela in the experiments can reliably sustain the scheduling of an input workload of approximately 100 req/s. While this is sufficient for our AR use case, Vela is capable of much higher SDPS, as we discuss in the next paragraph. The sudden increase in queuing time is due to the sampling threads waiting for responses from the Cluster Agents. A maximum CPU usage of 93% in the scheduler VM indicates that the current thread configuration is ideal and that the scheduler needs to be scaled out to further increase performance. Conversely, the Cluster Agents show a peak CPU usage of approximately 26%, indicating that thread-level parallelism could be further increased before scaling out, which we defer to future work.

The SDPS show a significant increase after 133 req/s, because of the number of rescheduling attempts, due to not finding suitable nodes. Rescheduling attempts rise from zero until 99 req/s and 0.04% at 133 req/s to 53.4% at 175 req/s and 75.75% at 290 req/s, resulting in up to 817 total SDPS in the last case. This is caused by resources becoming scarce in the cluster, which leads to not finding any suitable nodes during sampling. However, this shows that a single Vela Scheduler instance is capable of supporting high numbers of SDPS, while managing clusters of 20k total nodes.

As previously noted, scheduling conflicts are common in distributed schedulers. Their occurrence rate should be as low as possible to avoid rescheduling jobs, which consumes processing time. In Figure 6.9 we examine the percentage of scheduling conflicts of Vela Scheduler with and without the MultiBind mechanism. The number of scheduling conflicts with MultiBind is reported directly by our scheduler, while the number of conflicts without MultiBind is obtained by counting all successful commit phases, where MultiBind retried committing at least once. For the first five experiment iterations there are between zero and two scheduling conflicts with MultiBind. Then, the rate starts increasing gradually, but stays below 1% of the total jobs until a throughput of 94 jobs/s, reaching its highest value of 2.76% at 107 jobs/s. These numbers are very low compared to scheduling without MultiBind. In this case there are 2.09% scheduling conflicts already in the first experiment iteration, gradually increasing up to a maximum of with 28.2% at 107 jobs/s. This clearly shows the benefit of MultiBind; without it, the scheduling time would double or triple for up to 25% of the jobs, because they would need to traverse the Vela Scheduler lifecycle two or three times, due to rescheduling. Altogether the numbers show very promising results, with Vela Scheduler having linear scalability and the MultiBind mechanism being a great improvement over a simple rescheduling on conflict approach.

6.5 Summary

In this chapter we have presented Vela, a globally distributed, orchestrator-independent scheduler for the Edge-Cloud continuum with a 3-phase scheduling workflow. The first phase consists of the 2-Smart Sampling mechanism, which delegates sampling to globally distributed clusters, freeing the scheduler from communicating with nodes directly, thus, reducing latency. By considering the requirements of a job during sampling, the clusters produce meaningful samples that only contain nodes that are capable of hosting the job, thus reducing sample wastage and keeping the sample size small. The second phase filters and scores the sampled nodes according to multiple criteria and selects the best three nodes as candidates to host the job. The third and final phase commits the job to a node using the MultiBind mechanism. MultiBind greatly reduces scheduling conflicts by retrying committing a job to the second or third best suitable node, if the assignment to a previous node fails, which significantly increases scheduler throughput. We have evaluated Vela Scheduler on a testbed with 10 clusters with up to 20k simulated nodes, showing its scalability.

CHAPTER 7

ChunkFunc: Dynamic SLO-aware Configuration of Serverless Functions

Cost-efficient use of serverless workflows requires selecting resource configurations for each function that ensure that the end-to-end response time SLO of the workflow is met, while not spending more money than necessary. This task is further complicated by the fact that different function inputs may require different processing times. The ChunkFunc resource configuration optimizer automatically profiles individual functions with typical inputs to create input size-specific performance profiles. These profiles are leveraged to dynamically optimize the resource configuration of every function during the execution of a workflow with the goal of fulfilling the workflow's response time SLO and keeping the costs at a minimum.

7.1 Introduction

All major Cloud platforms provide serverless offerings [10, 151, 89, 107] and their usage is continuously growing. In a 2023 survey, Datadog reports that over 70% of its AWS customers and 60% of its Google Cloud customers use at least one serverless solution [48]. Serverless computing provides the advantage that developers can focus on the business logic of their functions and leave scaling and most infrastructure management decisions to the Cloud provider. Typically, developers only configure the amount of memory that should be allocated to a function. The memory maps to a predefined resource profile, which contains a fixed amount of vCPUs – we adopt the same convention for our work.

This chapter is based on the paper T. Pusztai and S. Nastic, "ChunkFunc: Dynamic SLO-aware Configuration of Serverless Functions," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

Despite the seeming simplicity of configuring serverless resources, tuning the amount of memory, vCPUs, and configuration models to ensure that SLOs are met, while minimizing the costs still remains a challenge [134, 132].

7.1.1 Tuning of Serverless Workflow Configurations

Tuning resource configurations of serverless workflows to meet SLOs is typically done using performance models for the comprising functions. There are two main types of approaches: 1) a-priori profiling of functions to build a performance model in an offline fashion and 2) monitoring of function executions at runtime to build the performance model in an online fashion.

1. A-priori profiling systems normally execute functions under varying resource configurations with typical input data to build a performance profile. This is used to configure the function's resources in production to meet the defined SLO. Most systems that tune entire workflows rely on graph algorithms [65, 135, 254]. Another approach is the use of a max-heap [204]. For a single function or job, linear, binary, and gradient descent search [270], Bayesian Optimization (BO) [5], and CPU time accounting [44] have been used. Two common drawbacks of a-priori profiling systems are that a "typical workload" needs to be defined and the tedious profiling process itself. Finding a typical workload might not be possible for functions that have highly variable inputs, such as those used for log or video processing. Profiling often needs to be done manually and/or takes a long time if all resource profiles need to be tested exhaustively. Some approaches reduce the number of profiling runs, e.g., using BO, but they require manual tuning of parameters to get accurate results.

2. Systems that build a performance model online rely on historical or live monitoring data of function executions. Some approaches passively monitor execution [62, 9, 270]. Others assign different configurations until the performance model is complete [198, 259], often relying on statistical methods, such as Bayesian Optimization, to reduce the number of configurations that need to be explored. However, until the performance model is complete, these approaches may violate the SLO. Thus, to have good SLO adherence from the first day in production, developers need to issue many requests to allow the model to train, which is essentially similar to profiling. Additionally, resources for collecting and processing monitoring data during the entire application lifetime to update the performance model may incur additional costs.

7.1.2 Motivation

Current approaches often overprovision resources [215, 238] and do not account for input data size variations, which leads to problems with highly heterogeneous workloads, because different input sizes may result in different performance under various resource configurations [111]. A common use case that deals with varying input data sizes is logs processing, e.g., hourly logs processing of a bank gets more data during the day than

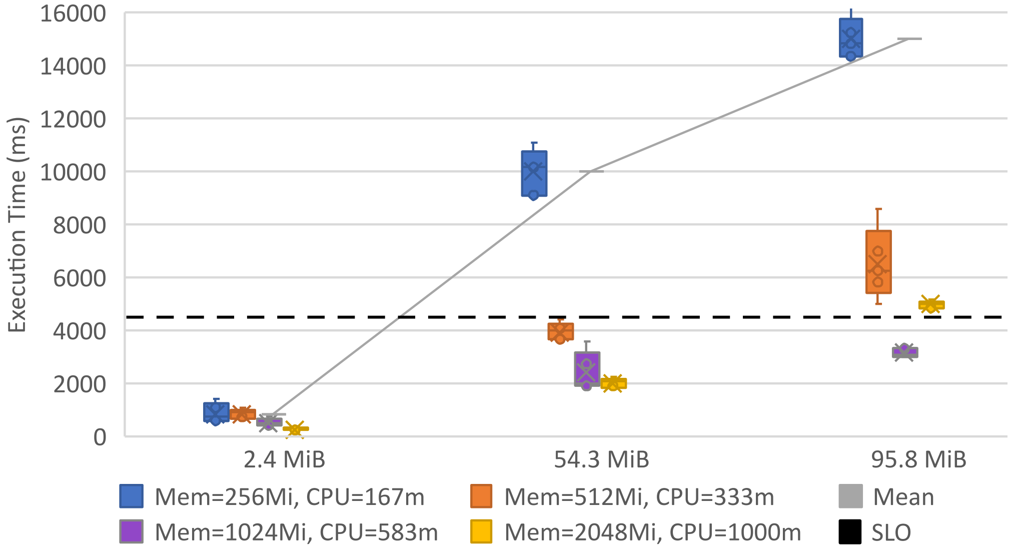


Figure 7.1: `extract-successes` Response Times under Various Input Data Sizes and Resource Configurations.

at night. Other examples include video processing (varying lengths, resolutions, and bitrates), malware scanning (varying file sizes), and continuous integration workflows in software engineering (varying repository sizes).

To further explore the need for input data size awareness, we run an experiment with a serverless function on Knative[1]. The `extract-successes` function extracts success messages from logs of a real distributed cluster scheduler [187]. We feed three log sizes (2.4, 54.3, and 95.8 MiB) to the function under four resource configurations (256, 512, 1024, and 2048 MiB) and measure the response time within the function itself, i.e., it is not affected by cold starts. Each combination is executed five times, the results are shown in Figure 7.1.

We define a *maximum response time (MRT)* SLO of 4,500 ms, indicated by the black dashed line in Figure 7.1. We observe that the mean response time across all resource configurations (gray line) increases with the input data size. Furthermore, we see that for meeting the SLO, different resource configurations may be used for different input data sizes. For the smallest input of 2.4 MiB, all four configurations meet the SLO, so the lowest (and cheapest) resource configuration with 256 MiB memory is sufficient. For the medium input of 54.3 MiB, only three configurations meet the SLO, with the lowest possible being 512 MiB memory. For the large input of 95.8 MiB, only the 1024 MiB configuration, meets the SLO, while the highest configuration violates the SLO. In theory, however, the single-threaded `extract-successes` Node.JS function should perform best with at least one CPU core, i.e., the 2048 MiB configuration or higher. Further investigation with a single resource-constrained Docker container showed that

[1] `https://knative.dev`

this behavior is specific to running the function in our target Kubernetes cluster and is caused by an interplay of the execution environment, the Node.JS IO thread pool, and the function structure. The automatic profiling results in our later experiments confirm that in the target cluster the 1024 MiB configuration is the fastest for the 95.8 MiB input.

Preliminary findings. The initial experiments show two important correlations for many serverless functions: i) when the input data size increases, the response time increases too and ii) for *a given input data size, a different resource configuration may increase or decrease the response time.* Consequently, there is usually not a single resource configuration that is ideal to meet a function's SLO and minimize its cost, but different resource configurations, depending on the input data size of an invocation. While there are exceptions, e.g., image labeling with almost constant runtime, many applications, like the previous examples, exhibit these correlations and, thus, benefit from input data size-aware resource configuration.

Shortcomings of the state-of-the-art. Most existing systems have at least one of two major shortcomings: i) they do not consider the size of the input data when choosing a resource profile for a function and/or ii) building the performance model for a function is a tedious, long profiling process or requires observing the live system for a long time. Most systems disregard the input data size when assigning a resource profile to a function, e.g., [65, 135, 204, 131, 198, 254]. This can result in SLO violations if a production input is substantially larger than the one(s) used for building the performance model and in excessive costs if the input is smaller than expected by the model. Building the performance model through profiling or by observing the live system requires time. Some approaches try to shorten that time, e.g., using Bayesian Optimization [269, 198] or regression [62] to reduce the amount of observations needed to build the performance model. However, to the best of our knowledge, none of these approaches account for different input sizes. With the contributions of this chapter, we address both of these shortcomings.

7.1.3 Contributions

In this chapter, we present ChunkFunc, a framework that dynamically adapts resource configurations of serverless functions, based on their input data size (payload) and reduces costs, while ensuring that the SLOs of the entire workflow are met. ChunkFunc is part of Polaris SLO Cloud[2], a SIG of the Linux Foundation Centaurus project[3], a platform for building unified and highly scalable distributed Cloud and Edge systems. Specifically, the main contributions include:

1. **An SLO- and input data size-aware function performance model** for determining optimized configurations in serverless workflows, depending on the input data size (Section 7.2).

[2] https://polaris-slo-cloud.github.io

[3] https://www.centaurusinfra.io

2. ***ChunkFunc Profiler*, which automatically builds performance models for serverless functions and workflows** based on typical input data sizes. Profiling is automatic, users only deploy a function and specify typical input data. A novel, auto-tuned BO approach reduces the profiling costs by up to 90% compared to exhaustive profiling and ensures high accuracy of the results. Contrary to state-of-the-art BO approaches we reuse the Gaussian Process (GP) of the BO to infer missing parts of our performance model (Section 7.3).

3. ***ChunkFunc Workflow Optimizer*, which leverages various heuristics to dynamically adapt the resource configuration of functions in a workflow to meet a performance-based SLO** (e.g., response time), while minimizing cost. Unlike existing systems, the ChunkFunc Workflow Optimizer considers the size of a function's input when selecting a resource profile, which, depending on the workflow, increases SLO adherence by a factor of 1.04 to 2.78 and reduces costs by up to 61% The Workflow Optimizer is extensible with arbitrary performance-based SLOs (Section 7.4).

The remainder of this chapter is structured as follows: Section 7.2 formulates the optimization problem, Section 7.3 presents the ChunkFunc Profiler, Section 7.4 describes the ChunkFunc Workflow Optimizer, in Section 7.5 and Section 7.6 we evaluate ChunkFunc by comparing it to two state-of-the-art solutions, and Section 7.7 summarizes the chapter.

7.2 ChunkFunc System Model & Optimization Problem

7.2.1 ChunkFunc System Model

A serverless workflow consists of functions chained together in sequence, in parallel, or in a combination of both, can be represented as a DAG $W = G(F, E)$. The set of nodes consists of the functions of the workflow, i.e., $F = \{f_0, f_1, \ldots, f_n\}$, and the set of edges $E = \{(f_i, f_j), \ldots\}$ consists of the invocation relationships among those functions. A directed edge (f_i, f_j) indicates that f_j is invoked with the output of f_i. The input to a function f_i is denoted as x_i and its size as $|x_i|$. The size of the output $f_i(x_i)$ depends on the particular function and, typically, it cannot be determined from the input data size without executing the function. The same input data size may yield different output sizes, e.g., the output size of a function that extracts error messages from a 1 GB log file depends on how many error messages the file contains.

Each function instance is assigned a set of resources, such as CPU and memory, which are defined in a resource profile p. The set RP contains all resource profiles that are available on the underlying serverless platform. Typically, commercial Cloud providers, allow users to only choose the amount of memory that should be assigned – each memory size is associated with a predefined number of CPU cores or fraction of CPU cores [90, 41]. We denote an instance of function f deployed with resource profile p as f^p. As noted in Section 7.1.2, serverless functions often exhibit a different performance for different

Table 7.1: Symbols Used in the System Model

Symbol	***Definition***
f	Serverless Function
p	Resource Profile
RP	All resource profiles p that are available on the underlying serverless platform
f^p	f deployed with resource profile p
$\|x\|$	Size of input data x
$M_{SLO}\left(f^p, x\right)$	SLO metric value of f^p, when executed with input x
$C\left(f^p, x\right)$	Cost of executing f^p with input x
PP_f	Performance profiles for f $PP_f = \bigcup_{\forall x \in X_f} \left\{\left(M_{SLO}\left(f^p, x\right), C\left(f^p, x\right)\right)\right\}$
$W = G\left(F, E\right)$	Workflow DAG with $F = \{f_0, f_1, \ldots, f_n\}$ $E = \{(f_i, f_j), \ldots\}$
s_W	SLO for the entire workflow W
RP_W	Selected resource profiles $\forall f \in W$
X_W	Set of input sizes $\forall f \in W$
$M\left(W, RP_W, X_W\right)$	SLO metric value for executing W with inputs X_W under resource profiles RP_W
$C\left(W, RP_W, X_W\right)$	Cost of executing W with inputs X_W under resource profiles RP_W

resource profiles. Thus, we denote the SLO metric of f^p, when executed with input x as $M_{SLO}\left(f^p, x\right)$. This metric can be the response time or another metric that corresponds to the desired SLO. Each resource profile has a cost associated per unit of execution time. $C\left(f^p, x\right)$ expresses the cost incurred by executing f with input x, when it is deployed with resource profile p. We observe that, given a function f to be invoked with input x, the SLO metric value and cost of this invocation depend on the chosen resource profile p. The pair $\left(M_{SLO}\left(f^p, x\right), C\left(f^p, x\right)\right)$ constitutes a performance profile for f under the resource profile p. The performance profiles for all typical inputs for f are collected in the set PP_f.

In addition to functions, a complex workflow may contain branch statements or loops. For the sake of simplicity, we consider these constructs also as functions within our optimization problem, albeit with special properties. Branch functions always have an SLO metric and cost of zero and loop functions wrap another function. The SLO metric value and cost of the loop function is equal to that of the wrapped function, multiplied by the number of loop iterations, which is known only at runtime. We denote the SLO of the entire workflow W as s_W.

For clarity, all symbols used in the system model and the optimization problem are summarized in Table 7.1.

7.2.2 Optimization Problem

The ChunkFunc optimization problem aims to find a set of resource profiles to deploy the functions of a workflow, given a particular input, while ensuring that the SLO is met and the cost of the workflow execution is minimized.

We use $RP_W = \{(f_0, p_0), (f_1, p_1), \ldots, (f_n, p_n)\}$ to denote the set of resource profiles that have been chosen to spawn the function instances in a particular execution of W, such that p_i is used to spawn f_i.

Let $X_W = \{|x_0|, |x_1|, \ldots, |x_n|\}$ be the set of input sizes to the functions of an execution of W, such that x_i is the input to f_i. The only element of X_W that is known at the beginning of the workflow is the input to the first function x_0; the remaining elements are added as the workflow progresses.

To enforce the SLO for a workflow W, we need the SLO metric value of a particular workflow execution, given the set of chosen resource profiles RP_W and function inputs X_W. It is calculated by aggregating all function SLO metric values:

$$M_{SLO}\left(W, RP_W, X_W\right) = \bigtriangleup_{f_i \in W} M_{SLO}\left(f_i^{p_i}, x_i\right) \tag{7.1}$$

Based on the type of SLO metric the semantics of the aggregation operator $\bigtriangleup$ change. There are two types of SLO metrics: i) *additive metrics*, such as response time, which are summed along a path in the workflow ($\bigtriangleup = \sum$) and ii) *min-metrics*, such as throughput, where the minimum of all edges in a path is taken ($\bigtriangleup = \min$).

We compute the total cost of the workflow execution, by summing the costs of its function executions:

$$C\left(W, RP_W, X_W\right) = \sum_{f_i \in W} C\left(f_i^{p_i}, x_i\right) \tag{7.2}$$

The optimization problem consists in finding a set $RP_W \subset F \times RP$ for an input set X_W that fulfills constraints (7.3) and (7.4). The former is a hard constraint and establishes the relation between the SLO metric value of the workflow and the SLO s_W. Depending on the type of SLO, $\lessdot$ is typically either $\leq$ (e.g., for response time) or $\geq$ (e.g., for throughput). The latter is a soft constraint that seeks to minimize the total cost of the workflow execution.

$$M\left(W, RP_W, X_W\right) \lessdot s_W \tag{7.3}$$

$$\min C\left(W, RP_W, X_W\right) \tag{7.4}$$

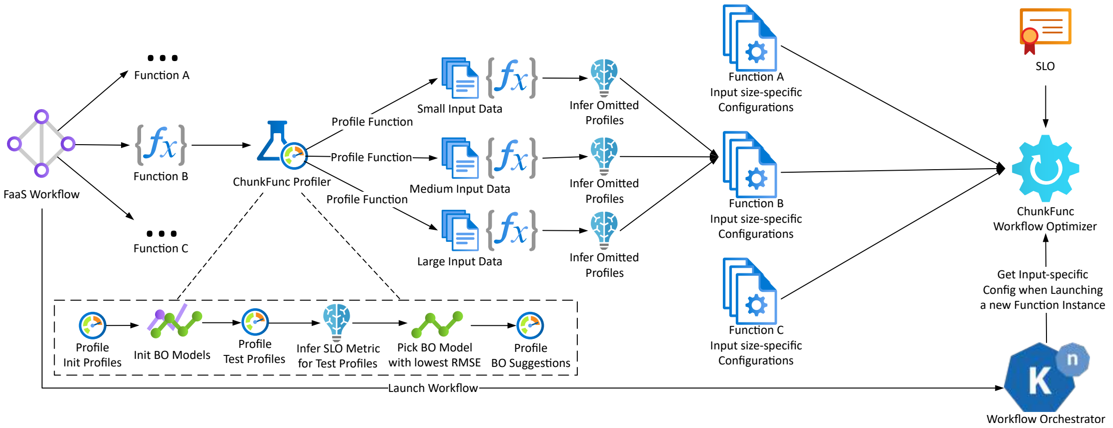


Figure 7.2: Overview of the ChunkFunc System and Lifecycle of a Serverless Workflow.

This optimization problem is NP-hard and the fact that only the first input size is known at the beginning of the workflow execution further complicates finding a solution. Any function's input, other than f_0's, is only known once all of its immediate predecessors have executed.

For example, consider a simple, sequential workflow with two functions, f_0 and f_1, and an MRT SLO of $s_W = 80$ ms. For the input data size $|x_0|$ the function f_0 takes 50 ms when deployed with the cheap resource profile p_0 and 25 ms when deployed with the expensive resource profile p_0'. The output of f_0 will either be small (x_s) or large (x_l). For a small input x_s the function f_1 takes at most 20 ms, while for a large input x_l, it takes at least 40 ms. Thus, when selecting a resource profile for f_0, the circumstance that we do not know the size of its output does not allow us to select the cheap resource profile with an execution time of 50 ms, because if the output happens to be large, f_1 will run for at least 40 ms, leading to a total response time of 90 ms, which violates the SLO. Since elements of X_W, are missing when the workflow is invoked, we cannot find an exact solution to the optimization problem at this point. However, we can approximate a solution using a heuristic, which we describe in Section 7.4.

7.3 ChunkFunc Framework Overview & Profiler

The ChunkFunc framework consists of two major components: the Profiler and the Workflow Optimizer. In this section, we first present an overview of the system and then describe the Profiler.

7.3.1 Framework Overview

Figure 7.2 presents an overview of ChunkFunc and the lifecycle of a serverless workflow within the system. Upon their deployment, serverless functions are automatically picked up by the ChunkFunc Profiler. It deploys function instances using various resource configurations to execute profiling runs with their typical input data sizes, without any user interaction. To reduce the number of profiling runs, while maintaining a high accuracy of the results, the choice of resource configurations is guided by Bayesian Optimization. Our BO Dynamic Hyperparameter Selection picks the hyperparameter that yields the most accurate results for a particular function type and input size combination. Finally, the input-specific performance profiles are leveraged by the ChunkFunc Workflow Optimizer, which provides a suitable resource profile, to meet the workflow's SLO and minimize cost, to the serverless orchestrator prior to invoking a function.

7.3.2 ChunkFunc Bayesian Optimization-based Profiler

The ChunkFunc Profiler automatically creates input data size-specific performance profiles for every deployed serverless function. The user only needs to specify several typical input data payloads (normally two or three) of different sizes for the function as ChunkFunc-specific metadata. For each defined typical input data size, a distinct performance profile is computed fully automatically by ChunkFunc.

While exhaustively profiling the function under every resource profile is supported, it can incur high costs. Thus, we leverage Bayesian Optimization (BO) to reduce the number of profiling runs. BO is a technique that is normally used to find the maximum of an unknown objective function, based on a limited set of samples [126]. It builds a surrogate model, typically using a Gaussian Process, to approximate the objective function using known samples and uses an acquisition function to guide the exploration of further samples.

BO is deeply integrated into the ChunkFunc profiler in two ways. First, we use the acquisition function to determine the most promising input data sizes to profile, similar to [269, 198]. Second, we leverage the surrogate model to infer the SLO metric for those input data sizes that were not profiled. This use of BO to infer the missing parts of the performance profile has, to the best of our knowledge, not been attempted by the state-of-the-art. Our BO-based profiler allows for an up to 90% reduction of profiling costs.

Common choices for the acquisition function of BO include Probability of Improvement (POI) [126] and Expected Improvement (EI) [154, 117]. POI returns the probability that sampling a certain point will yield an improvement, but it may easily result in focusing only on a specific region of the objective function (exploitation) or jump around too much (exploration). EI aims to quantify the improvement and is less prone to the aforementioned issues [116]. In ChunkFunc we rely on both: we use EI to determine which point, i.e., resource profile p, to profile next and POI to define the stopping criterion. Since EI yields an absolute value and POI a percentage, the latter is more suitable as a stopping criterion.

Our aim is to achieve a relative root mean square error (RMSE) of 10% or less when comparing the BO-guided profiling results for an input data size to exhaustive profiling results to ensure that the inferred profiling results adequately reflect the actual performance. We use two stopping criteria for BO: i) the POI for sampling the next resource profile is below 2%, provided that we have sampled at least 10% of the available resource profiles, or ii) we have sampled 40% of the available resource profiles. Based on our experience it is necessary to sample at least 10% of all resource profiles, because for some functions the POI is already below 2% after the initial samples, but the relative RMSE would be above 10%. The second stopping criterion is necessary, because for some functions the POI does not drop below 2%, even though the RMSE is already sufficiently low.

Each input data size $|x|$, resulting from the user-defined discrete set of typical inputs, is profiled independently with a distinct BO model. Once a stopping criterion is fulfilled, the performance profile of the function with the input data size $|x|$ is built. For each resource profile the profiler takes either the mean SLO metric that was measured, if the resource profile has been evaluated, or uses the BO's surrogate model (GP) to infer the SLO metric. The number of inputs needed for profiling varies depending on the function. Determining this number is beyond the scope of this work, but we will outline a solution in Section 7.6.3.

7.3.3 Bayesian Optimization Hyperparameter Selection

The key to make the ChunkFunc Profiler converge quickly to an accurate solution is to pick the acquisition function's hyperparameters correctly – for EI this is the ξ hyperparameter. ξ determines whether the Profiler's acquisition function will favor exploring unknown ranges of the input domain to find this maximum (higher ξ values) or focus on finding the maximum in an already known range (lower ξ values). Since we could not observe any correlation pattern between function type, input data size, and the ξ value that yields the lowest RMSE, we devised a dynamic hyperparameter selection approach, which we describe in Algorithm 7.1.

We start by selecting a fixed number of resource profiles RP_{init}, evenly distributed from the set of all resource profiles, and profiling the serverless function with each of them. For each resource profile a function instance is deployed, executed once to avoid cold starts, and then executed five times to obtain mean measurements for the SLO metric and the cost. These measurements are used to initialize one BO model for each of the candidate hyperparameter ξ values.

Next, we pick another set of evenly distributed profiles RP_{test}, which will be used to test the accuracy of the BO models. We profile the function with these to get SLO metric measurements M_{test} to compare the predictions against. For each BO model we infer the SLO metric values for the profiles in RP_{test} and, then, compute the RMSE to the actual measurements M_{test}.

Finally, we pick the BO model that yields the lowest RMSE, add M_{test} to it, and continue profiling with it until one of the stopping criteria is fulfilled. This allows us to select the most suitable ξ hyperparameter without additional profiling runs in the average case. In

Algorithm 7.1 Bayesian Optimization Dynamic Hyperparameter Selection.

```
Input: $f$: Function to be profiled;
    $x$: Input to be profiled;
    $\Xi$: Set of $\xi$ candidate values;
    $RP_{all}$: Set of available platform resource profiles;
Output: $BO_f^x$: Initialized BO model for $f$ with input $x$;

$RP_{init} \leftarrow \textsc{GetInitProfiles}(RP_{all})$
$M_{init} \leftarrow \textsc{RunProfiling}(f, x, RP_{init})$
$BO \leftarrow \{\ \forall \xi \in \Xi : \textsc{NewBOModel}(\xi, M_{init})\ \}$

$RP_{test} \leftarrow \textsc{GetTestProfiles}(RP_{all})$
$M_{test} \leftarrow \textsc{RunProfiling}(f, x, RP_{test})$
$RMSE \leftarrow \{\}$
for all $BO_\xi \in BO$ do
    $M_{inf} \leftarrow \{\}$
    for all $p \in RP_{test}$ do
        $M_{inf} \leftarrow M_{inf} \cup \{\ \textsc{Infer}(BO_x i, p)\ \}$
    end for
    $RMSE \leftarrow RMSE \cup \{(BO_\xi, \textsc{CalcRMSE}(M_{inf}, M_{test}))\}$
end for

$(BO_f^x, rmse) \leftarrow \textsc{FindLowestRMSE}(RMSE)$
$\textsc{AddSamples}(BO_f^x, M_{test})$
return $BO_f^x$

function $\textsc{RunProfiling}(f, x, RP)$
    $M_{RP} \leftarrow \{\}$                                    ▷ SLO metric measurements
    for all $p \in RP$ do
        $(m_x, c_x) \leftarrow \textsc{ProfileWithInput}(f, p, x)$
        $M_{RP} \leftarrow M_{RP} \cup \{(p, m_x)\}$
    end for
    return $M_{RP}$
end function

function $\textsc{ProfileWithInput}(f, p, x)$
    $f^p \leftarrow \textsc{Spawn}(f, p)$                           ▷ Spawn $f$ with profile $p$
    $f^p(x)$                                              ▷ Invoke $f^p$ to avoid cold start
    $M_x \leftarrow \{\}$                                   ▷ SLO metric measurements for $x$
    for $i \leftarrow 0$ to $numIterations$ do
        $f^p(x)$
        $M_x \leftarrow M_x \cup \{\ \textsc{GetSloMetric}(\ )\ \}$
    end for
    $\textsc{Destroy}(f^p)$
    $(m_x, c_x) \leftarrow \textsc{CalcMeanSloMetricAndCost}(M_x)$
    return $(m_x, c_x)$
end function
```

the worst case, if a stopping criterion would be met after 10% of the resource profiles, obtaining M_{test} results results in a negligible number of additional profiling runs.

7.4 ChunkFunc Workflow Optimizer

ChunkFunc Workflow Optimizer leverages the performance profiles to assign resource profiles to each individual function instance in a workflow, based on the input data sizes, while fulfilling the SLO and minimizing cost. The SLO serves as an upper or lower bound for the aggregated SLO metric of the entire workflow, while the total cost should be minimized. Since the set of function inputs X_W is filled step by step as the workflow executes, we need a heuristic to approximate the solution of the ChunkFunc optimization problem as the workflow progresses.

Before executing a function, the workflow orchestrator queries the Workflow Optimizer for the resource profile. Akin to the optimization problem, the Workflow Optimizer models the workflow as a DAG. To determine the resource profile for a function, the Workflow Optimizer needs its input data size. The heuristic receives as input the workflow graph, the SLO, the input data size for the current function, and the SLO metric value for the current execution path. Since the heuristic is invoked for each node, while the workflow is executing, it can react if previous functions affected the SLO metric differently than expected, e.g., they took more time than expected.

The *Proportional Critical Path heuristic* can use any performance metric as SLO metric. This heuristic derives a sub-SLO for the current function and chooses the cheapest resource profile that allows meeting the sub-SLO, based on the function's performance profiles. Adapting the heuristic for cost-based SLOs is possible and planned as future work.

Since metrics of functions may vary between workflow invocations, for any function, other than the first one, the remaining SLO metric until a violation of the workflow SLO may differ. For example, for an MRT SLO suppose $s_W = 100$ ms, if f_0 takes 10 ms, the remaining time for f_1 and its successors is 90 ms. If f_0 took 15 ms, the remaining time would be 85 ms. Thus, the each function's sub-SLO must be calculated dynamically before selecting a resource profile.

To compute the sub-SLO of a function f_i, we need to know how much it contributes to the overall SLO metric of the remaining workflow. The latter is the length of the critical path from (including) f_i until the end of the workflow. We define the critical path as the longest path between two nodes [127], with an edge's weight being the SLO metric of its target node. Since many metrics vary depending on input data sizes, finding the critical path is not trivial. We compute the mean SLO metric value of every function across all resource profiles and input data sizes and use these values as weights for the critical path. If the SLO metric is an additive metric, we now add the mean SLO metric value of f_i to the critical path to allow us to calculate f_i's proportional contribution to it. This proportion used on the remaining workflow SLO yields the sub-SLO. For a min-metric, we take the minimum of the f_i's SLO metric value and the aggregated SLO metric value of the critical path. Algorithm 7.2 outlines the Proportional Critical Path heuristic:

Step 1. Line 3 computes the sub-SLO of function f using `ComputeSubSLO()`. For

Algorithm 7.2 Proportional Critical Path Heuristic.

```
Input: $f$: Function, for which to select the resource profile;
    $x$: Input for $f$;
    $PP_f$: Set of performance profiles for $f$;
    $W = G(F, E)$: Workflow DAG;
    $s_W$: SLO for $W$;
    $M_{avg}$: Mean SLO metrics for all functions in $W$;
    $m_{curr}$: Current SLO metric value, e.g., elapsed time for MRT SLO;
Output: $p$: Selected resource profile for $f$;

$s_f \leftarrow$ ComputeSubSLO($f, s_w, m_{curr}, M_{avg}$)
if $PP_f$ contain GP inferences then
    $s_f \leftarrow s_f * safetyMargin$
end if
$p \leftarrow nil$                                          ▷ The selected resource profile
$m_p \leftarrow \infty$                                     ▷ SLO metric value under profile p
$c_p \leftarrow \infty$                                     ▷ Cost under profile p
$PP_f^x \leftarrow$ GetPerfProfilesForInputSize($PP_f, |x|$)
for all $pp_i \in PP_f^x$ do
    $m_f \leftarrow$ GetSloMetric($pp_i$)
    $c_f \leftarrow$ GetCost($pp_i$)
    if $m_f \lessdot s_f$ then
        if $c_f < c_p$ OR ($c_f = c_p$ AND $m_f \lessdot m_p$) then
            $p \leftarrow$ GetResourceProfile($pp_i$)
            $m_p \leftarrow m_f$
            $c_p \leftarrow c_f$
        end if
    end if
end for
if $p = nil$ then
    $p \leftarrow$ GetFastestProfileForInputSize($PP_f, |x|$)
end if
return $p$

function ComputeSubSLO($f, s_w, m_{curr}, M_{avg}$)
    $end \leftarrow$ FinalNode($W$)
    $cp \leftarrow$ FindSloCriticalPath($f, end$)
    if $M$ is additive then
        $m_f \leftarrow$ GetAvgSloMetric($M_{avg}, f$)
        $m_{remaining} \leftarrow s_W - m_{curr}$
        $contrib_f \leftarrow \frac{m_f}{\text{GetSloMetric}(cp) + m_f}$
        return $m_{remaining} * contrib_f$
    else
        $m_{cp} \leftarrow$ GetSloMetric($cp$)
        return Min($s_w, m_{cp}$)
    end if
end function
```

additive SLO metrics we use Dijkstra's shortest path algorithm to find the critical path. The weight of each edge (f_i, f_j) is the negative mean SLO metric of f_j. All weights are negative, so Dijkstra's algorithm works normally and we get the longest path. For an additive metric we compute the sub-SLO using f's proportional contribution to the critical path, while for a min-metric we use the minimum of f's and the critical path's SLO metric values. After returning the sub-SLO we multiply it with a safety margin (line 5) if the performance profiles contain inferences from BO. This ensures that imprecisions resulting from the inferences do not affect SLO adherence.

Step 2. Lines 7–10 initialize the selected resource profile p to *nil* and f's SLO metric and cost to infinity. Then, f's performance profiles for the current input data size are retrieved. Performance profiles are stored in buckets, according to the input data size they were computed for. Input x matches the bucket with the smallest input data size that is greater than or equal to the size of x. For inputs that are greater than the largest bucket input data size, that greatest bucket is taken.

Step 3. Lines 11–21 iterate over f's performance profiles for the current input data size. For each performance profile pp_i we check if its SLO metric value allows meeting the sub-SLO (line 14). If that is the case, the cost of pp_i is examined (line 15). If pp_i is cheaper than the currently selected profile p or, if their costs are equal, but pp_i has a better SLO metric value, the selected profile is updated to the resource profile in pp_i.

Step 4. If no resource profile meets the sub-SLO, we fall back to the fastest profile for the input size, irrespective of its metrics, hoping that subsequent functions meet their SLOs. Finally, the selected resource profile is returned.

7.5 Implementation & Experiments Design

To evaluate ChunkFunc we focus on the quality of the Workflow Optimizer results, i.e., whether its resource profile selection meets the workflows' response time SLOs and how much the total cost is. We compare ChunkFunc to two state-of-the-art approaches. All code and data needed to run the experiments, as well as, additional results can be found in our repository[4].

7.5.1 Implementation

We implement ChunkFunc Profiler in Go as an open source[5] Kubernetes controller and target serverless functions realized with Knative. Without loss of generality, the Profiler currently triggers functions via HTTP requests, since this is a common and flexible invocation method. Our trigger mechanism abstraction allows for adding other trigger types, e.g., storage events, in the future. ChunkFunc-specific function metadata is passed

[4] `https://polaris-slo-cloud.github.io/chunk-func/experiments/`

[5] `https://github.com/polaris-slo-cloud/chunk-func` and `https://doi.org/10.5281/zenodo.14174081`

to the Profiler, using a Kubernetes CRD, i.e., a custom type of object that can be stored in the cluster. Each such `FunctionDescription` object contains a reference to the Knative function definition object and a list of typical inputs. Once the Profiler detects a new `FunctionDescription` it automatically starts profiling the referenced Knative function and, upon completion, adds the performance profiles to the `status` subresource of the `FunctionDescription`. To evaluate the Workflow Optimizer we design various workflows and for each we replay real-life function traces from our performance profiles in our custom simulator. Our simulation with real-life traces is deterministic, so it needs to be executed only once for each configuration, which enables faster exploration of a large range of SLOs.

7.5.2 Experiments Setup

To evaluate ChunkFunc we use three real-world and six synthetic serverless workflows. The real-world workflows are written for our research, but are similar to production use cases. They are i) a log processing workflow (*LogPro*), ii) a video processing workflow (*VidPro*), and iii) an ML-based face detection workflow (*FaceDet*). They represent typical examples of serverless workflows with variable input data size, while exhibiting different response time characteristics. LogPro takes a log file from a distributed cluster scheduler [187] from an S3-compatible storage bucket as input. The workflow consists of a sequence of four serverless functions that validate the log and extract various statistics. VidPro cuts out an unwanted segment of a video from S3, and encodes the rest in a predefined format for social media. The workflow consists of four functions that validate the video, cut and encode the two segments (two parallel instances of the same function), and merge the encoded segments. FaceDet detects and marks faces in a video from S3. It consists of a sequence of four functions: validation, transformation of the video to a standardized resolution, face detection, and marking of all faces in an output video.

Additionally, we use six synthetic workflows, which are assembled using profiling results from the real-world workflows. Like the real-world functions, the response time of the functions in the synthetic workflows is dependent on the input size, as determined by the profiling results. During generation of the workflows, each function's output is chosen from the set of supported input sizes of the successor function. For each workflow there are three input size configurations: small, medium, and large. The synthetic workflows are: i) *homogeneous*, a sequence of functions with the same (medium) resource requirements, ii) *LoHiRes*, a sequence of functions with low resource requirements, followed by a sequence of functions with high resource requirements, iii) *HiLoRes*, high resource functions, followed by low resource functions, iv) *random*, a random sequence of functions with low, medium, and high resource requirements, v) *cyclic*, a low resource function, followed by a medium resource, followed by a high resource function, repeated in cycles, and vi) *staircase*, a sequence of low resource functions, followed by a sequence of medium resource functions, followed by a sequence of high resource functions. The first four workflows consist of 40 functions each, while the last two consist of 42 functions.

Two sets of workflows allow us to demonstrate how ChunkFunc behaves with real-life applications and to use the longer and more complex synthetic workflows to evaluate ChunkFunc's scalability. Our workflows are mostly sequential, because ChunkFunc relies on the critical path in a workflow and even in a massively parallel workflow, the critical path is always sequential. The response times of our functions is within the range of the current state-of-the-art, e.g., AWS imposes a default function timeout of 3 seconds, which can be changed to a maximum of 15 minutes [12]. The average end-to-end durations (base SLOs) of our workflows cover a wide range, starting at 12 seconds for LogPro, approx. one minute for VidPro, 5 minutes for FaceDet, extending to 75 minutes for the synthetic HiLoRes workflow. The number of functions per workflow is representative of most serverless workflows currently in use. A large scale study [64] showed that 59% of workflows consisted of 2–10 functions, 19% of 10–1000 functions, and 3% more than that (19% could not be categorized).

We implement all real-world functions in TypeScript, except for face detection and marking, for which we use Python. We deploy them using Knative v1.10 on a Kubernetes v1.27 cluster. For video processing we wrap ffmpeg[6] v6.0 and use the x264[7] and AAC codecs. Face detection and marking relies on the OpenCV[8] library.

We run the experiments with two sets of resource profiles. The first set of profiles and their costs per 100 ms is coarse-grained and resembles the 128 MB – 16384 MB profiles available on Google Cloud Functions (GCF) [90] (Tier 2 prices). Since for GCF there are eight profiles in this memory range, we use exhaustive profiling for this set of resource profiles. The second set of resources profiles and their costs per 1 ms is fine-grained and resembles the 128 MB – 10240 MB range available on AWS Lambda [11]. Every memory size maps to the CPU core count defined by AWS [41]. AWS uses a continuous memory range, which we divide into 64 MB steps, which results in 159 resource profiles, for which we use BO-guided profiling.

We implement six heuristics: i) Fastest configuration, ii) Cheapest configuration, iii) ChunkFunc Proportional Critical Path heuristic (ChunkFunc), iv) ChunkFunc with known function output sizes (CF-Oracle), v) SLAM [204], and vi) StepConf [254]. CF-Oracle is identical to ChunkFunc, except that the former knows all function output sizes from an "oracle" when computing the critical path – this is only used in comparison to ChunkFunc to assess the effectiveness of function output size estimates compared to the actual output sizes when determining the critical path. Both heuristics compute an average across all resource profiles for the critical path. SLAM and StepConf both rely on offline profiling to build a performance model of the functions. We execute all experiments using the exhaustive profiling results and using the BO predicted profiling results.

SLAM precomputes all function configurations prior to executing the workflow. It inserts all functions using their response times under the lowest resource configuration into a max-heap. SLAM pops the slowest function off the heap, increases its resources to the

[6] https://www.ffmpeg.org
[7] https://www.videolan.org/developers/x264.html
[8] https://opencv.org

Table 7.2: Real-world Workflow Scenarios.

Workflow	**Input Sizes**	**SLO Interval (sec) for Profiles**	
		Coarse-grained	**Fine-grained**
LogPro	2.4 MiB 54.3 MiB 95.8 MiB	12.1s +/- 15% [10.3; 14.0]	-
VidPro	360p - 40 MiB 720p - 227 MiB 1080p - 500 MiB	77.4s +/- 35% [50.3; 104.5]	51.0s +/- 35% [33.2; 68.9]
FaceDet	20s, 720p - 6.51 MiB 40s, 720p - 26.5 MiB 60s, 1080p - 73.5 MiB	330s +/- 35% [214.5; 445.4]	302.2s +/- 35% [196.4; 408.0]

next higher profile, and reinserts it into the heap. If the resources cannot be increased further, the function's configuration is frozen, and it is not reinserted into the heap. SLAM continues until an SLO-compliant configuration is found or the heap is empty. A second version of the algorithm checks if the percentage of decrease in response time is greater than the percentage of cost increase before returning a function to the heap. We use the cheaper of the two results.

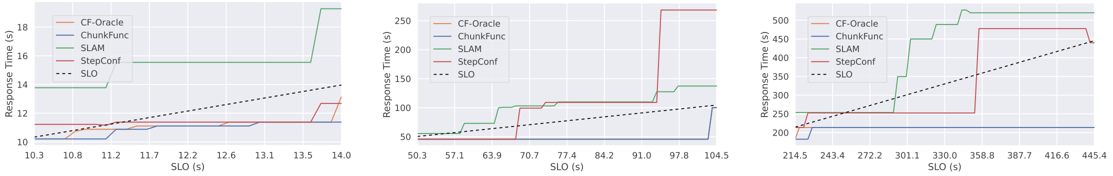
StepConf chooses each function's resource profile directly prior to its execution using an NP-hard algorithm or a heuristic on a DAG and is a representative for state-of-the-art graph-based algorithms. The heuristic we implemented for our experiments, computes a sub-SLO for each function step, based on its contribution to the critical path until the end of the workflow and the remaining time until SLO violation. For computing the critical path, the response time of the most cost-effective resource profile is used for every function.

Since SLAM and StepConf are unaware of different input data sizes, we use the profiling results for each function's median input data size for these strategies.

7.6 Experimental Results

7.6.1 Real-world Workflows

For each real-world workflow we create and profile scenarios with a small, a medium, and a large input size. To define MRT SLOs we use the fastest and the cheapest configurations as the lower and upper bounds. For example, for the largest input data size for VidPro the lower and upper bounds for the response time on the coarse-grained profiles are 44.788 seconds and 110.089 seconds. We define the $baseSlo = lowerBound + \frac{upperBound - lowerBound}{2}$, e.g., 77.4 s for VidPro. We explore the SLO interval of $baseSlo \pm N\%$ in one-percent steps, i.e., $N + 1$ distinct SLOs. We chose N s.t. the interval does not exceed the bounds given by the fastest and cheapest configurations.

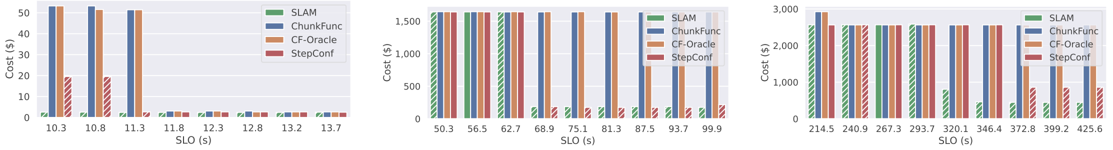


(a) LogPro 95.8 MiB Response Times (s)

(b) VidPro 500 MiB Response Times (s)

(c) FaceDet 73.5 MiB Response Times (s)

Figure 7.3: LogPro, VidPro, and FaceDet Maximum Response Time SLO Compliance for Large Inputs for Coarse-grained Resource Profiles.



(a) LogPro 95.8 MiB Costs per 10,000 Invocations

(b) VidPro 500 MiB Costs per 10,000 Invocations

(c) FaceDet 73.5 MiB Costs per 10,000 Invocations

Figure 7.4: LogPro, VidPro, and FaceDet Costs per 10,000 Invocations for Large Inputs for Coarse-grained Resource Profiles.

Since the available resources in the lowest and the highest profiles differ between the coarse-grained and the fine-grained resource profile sets, also the lower and upper response time bounds and, hence, the base SLOs differ. All workflow configuration scenarios are shown in Table 7.2.

Coarse-grained Resource Profiles - Exhaustive Profiling

Figure 7.3 shows the SLO compliance results as response time graphs for the large input data sizes (the other sizes are available in our repository). The dashed black line denotes the MRT SLO, i.e., to fulfill the SLO, the workflow's response time must be equal to or below this line. Table 7.3 shows details for all input data sizes.

All heuristics exhibit long periods of straight lines in the response time graphs, because they use a certain set resource configurations until the SLO relaxes enough to use a less powerful resource profile on one function – this behavior causes a straight line in the graph. Additionally, the relatively short workflows allow only few functions to be adapted, thus increasing the length of the straight lines; the synthetic workflows exhibit many more "steps" in the graphs.

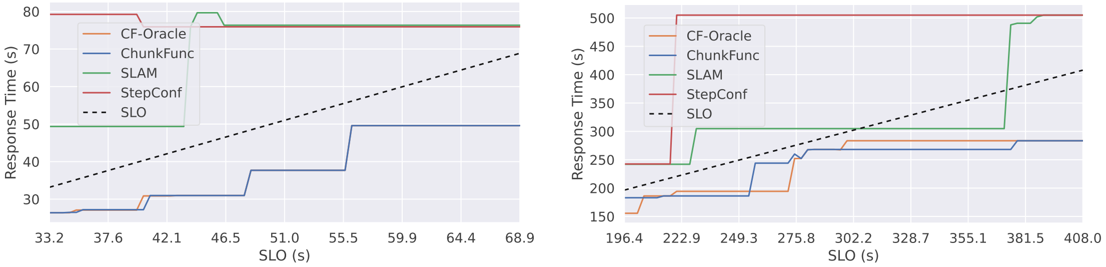
ChunkFunc (standard and CF-Oracle version) is the only heuristic that meets the SLO in all cases across all input sizes. SLAM and StepConf work well for one or two input sizes, but fail a substantial amount of SLOs in the rest. SLAM fulfills two thirds of the LogPro SLOs, 69% of VidPro, and 68% of FaceDet. SLO violations occur for medium and large inputs for FaceDet and only for large inputs for the other two. Compared to

Table 7.3: Real-world Workflows SLO Compliance for Coarse-grained Resource Profiles.

Workflow	Input Size	SLO Adherence				
		ChunkFunc	SLAM		StepConf	
LogPro	Small	100%	100%	67%	100%	91%
	Medium		100%		100%	
	Large		0%		74%	
VidPro	Small	100%	100%	69%	100%	78%
	Medium		100%		100%	
	Large		6%		34%	
FaceDet	Small	100%	100%	68%	100%	75%
	Medium		87%		73%	
	Large		17%		51%	
Overall		100%	68%		81%	

SLAM, ChunkFunc increases SLO adherence by 45% to 49%. StepConf fulfills 91% of the LogPro SLOs, 78% for VidPro, and 75% for FaceDet. Most violations occur for large input sizes, but for FaceDet StepConf also misses 27% of the SLOs for medium inputs. Compared to StepConf, ChunkFunc increases SLO adherence by 10% to 33%. Across all workflows, SLAM fulfills 68% of the SLOs, while StepConf meets 81%, this amounts to a mean increase in SLO adherence of 47% and 23% respectively, when using ChunkFunc instead.

Figure 7.4 shows the costs for 10,000 workflow invocations. If an algorithm violates an SLO the respective cost bar is shown with a hatch pattern, because if the SLO is not met, evaluating the cost is pointless. To ensure comparability we show the costs for each algorithm only where it meets the SLO. To avoid bias from SLO violating configurations, when analyzing the costs, we conduct a one-on-one comparison, where we consider only the cases where both strategies meet an SLO. We compare the mean costs of these cases. When comparing ChunkFunc to SLAM, ChunkFunc is 4% cheaper for LogPro, 54% cheaper for VidPro, and 19% cheaper for FaceDet. When comparing to StepConf, ChunkFunc is 165% more expensive for LogPro, 29% cheaper for VidPro, and 22% cheaper for FaceDet. For LogPro ChunkFunc is more expensive than StepConf for almost all SLOs (for some they are even). This is because ChunkFunc often picks faster resource profiles, because it knows that it needs to fulfill every sub-SLO for the large input, while StepConf assumes the medium input, for which the sub-SLO can be fulfilled with cheaper resource profiles. While this approach allows StepConf to save costs, it also causes it to miss the tight SLOs for large inputs. In the general case, ChunkFunc fulfills more SLOs than StepConf. In workflows with long-running functions, such as VidPro and FaceDet, ChunkFunc allows saving up to 48% of the costs over StepConf.



(a) VidPro 500 MiB Response Times.

(b) FaceDet 73.5 MiB Response Times.

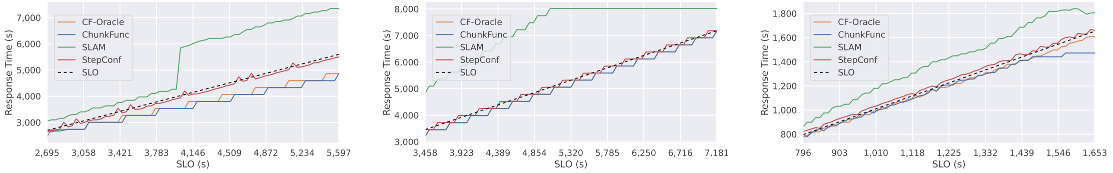
Figure 7.5: VidPro and FaceDet MRT SLO Compliance for Large Inputs for Fine-grained Profiles.

Fine-grained Resource Profiles - BO-guided Profiling

Figure 7.5a and Figure 7.5b show the SLO compliance results as response time graphs for the large input data sizes. Table 7.4 shows details for all input data sizes. The straight lines in the graphs are caused by the same reasons as for the coarse-grained resource profiles. ChunkFunc is the only heuristic that meets the SLO in all cases across all input sizes. Its SLO adherence is completely unaffected by whether we use the exhaustive profiling results or the BO-inferred profiling results. SLAM and StepConf work well for one or two input sizes, but fail a substantial amount of SLOs in the rest. SLAM fulfills two thirds of all VidPro SLOs and 74% of FaceDet, with SLO violations occurring mostly for large inputs. Compared to SLAM, ChunkFunc increases SLO adherence by 35% to 50%. StepConf fulfills 62% of the VidPro SLOs and 36% for FaceDet. Only for small input sizes all the SLOs are met, while as for medium input sizes there are already considerable violations for VidPro and almost entirely violated for FaceDet. Compared to StepConf, ChunkFunc increases SLO adherence by 61% to 178%. Across all workflows, SLAM fulfills 71% of the SLOs, while StepConf meets 49%, this amounts to a mean increase in SLO adherence of 41% and 104% respectively, when using ChunkFunc instead. We have excluded LogPro from the experiment with fine-grained resources profiles. This

Table 7.4: Real-world Workflows SLO Compliance for BO-inferred Fine-grained Profiles.

Workflow	**Input Size**	**SLO Adherence**				
		ChunkFunc	**SLAM**		**StepConf**	
VidPro	Small	100%	100%	67%	100%	62%
	Medium		100%		86%	
	Large		0%		0%	
FaceDet	Small	100%	100%	74%	100%	36%
	Medium		89%		7%	
	Large		32%		0%	
Overall		100%	71%		49%	



(a) Cyclic WF Large Input Response Times (s)

(b) HiLoRes WF Large Input Response Times (s)

(c) Homogeneous WF Large Input Response Times (s)

Figure 7.6: Representative Results of Synthetic Workflow Experiments for Coarse-grained Resource Profiles.

is because its functions are single-threaded (Node.JS) with low memory requirements. Since the fine-grained resource profiles all contain at least one vCPU, there is almost no performance difference between the resource profiles, hence the omission of LogPro from this experiment.

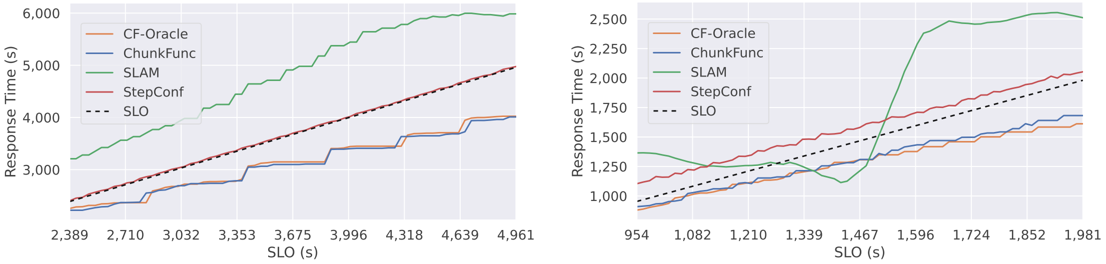
We do not show the cost graphs here, because SLAM and StepConf fail to meet almost all of the SLOs for large inputs. When comparing ChunkFunc to SLAM one-on-one across all input sizes, where both heuristics meet the SLO, ChunkFunc is 48% cheaper for VidPro and 6% more expensive for FaceDet. When comparing to StepConf, ChunkFunc is 36% more expensive for VidPro and 42% more expensive for FaceDet. However, ChunkFunc fulfills many more SLOs than SLAM and StepConf. This justifies a slight increase in cost for one workflow with respect to SLAM. With respect to StepConf, the cost increases are more substantial. However, these increases cover less than two thirds of the SLOs for VidPro and only slightly over one third of the SLOs for FaceDet; for the remainder StepConf fails to meet the SLO.

7.6.2 Synthetic Workflows

Coarse-grained Resource Profiles - Exhaustive Profiling

The synthetic workflows are used to evaluate ChunkFunc's scalability in longer, more complex workflows. The homogeneous, LoHiRes, HiLoRes, and random workflows consist of 40 functions in sequence. The cyclic and staircase workflows use a short-running, medium-running, and a long-running function, each of which appears 14 times in the workflow, hence they consist of a total of 42 functions. For all synthetic workflows we simulate scenarios with a small, a medium, and a large input.

Figure 7.6 shows the SLO adherence for the coarse-grained profiles for the large inputs to the cyclic, HiLoRes, and homogeneous workflows, which we use as a representative examples (for other graphs please see our repository). The SLO adherence of the heuristics shows three pattern categories: For the cyclic, random, and staircase workflows the heuristics exhibit the pattern exemplified in Figure 7.6a. The LoHiRes and HiLoRes workflows show the pattern in Figure 7.6b. The SLO adherence for homogeneous workflow has its own distinct pattern shown in Figure 7.6c.



(a) Cyclic WF Large Input Response Times (b) Homog. WF Large Input Response Times

Figure 7.7: Representative Results of Synthetic Workflow Experiments for Fine-grained Resource Profiles.

ChunkFunc's pattern shows only minor differences between the workflows. It is the only heuristic that meets all SLOs for all input sizes. StepConf's pattern remains consistent across all workflows. For large inputs, it varies closely between fulfilling and violating the SLOs. Across all six synthetic workflows and three input sizes, it meets 79% of the SLOs, with the lowest value being 60% for the homogeneous workflow and the highest being 96% for the cyclic workflow. SLAM exhibits the largest differences in its patterns. It violates all large input SLOs, but fulfills all SLOs for the other inputs, yielding an average adherence of 67%. For the cyclic, random, and staircase workflows, SLAM's response times are first close to the SLO line and diverge at some point from it. For HiLoRes and LoHiRes the response times are always far from the SLO until they plateau out at some point. For the homogeneous workflow, SLAM's response times are closer to the MRT SLO line. For ChunkFunc the results yield an increase in SLO adherence of 27% over StepConf and 50% over SLAM.

For costs we perform the same one-on-one comparison for fulfilled SLOs that we did for the real-world workflows. For the cyclic workflow ChunkFunc is 48% cheaper than SLAM and 27% cheaper than StepConf. For the staircase workflow ChunkFunc only requires 39% of the costs of SLAM, making it 61% cheaper. On average ChunkFunc is 38% cheaper than SLAM and 10% cheaper than StepConf.

Fine-grained Resource Profiles - BO-guided Profiling

Figure 7.7 shows the SLO adherence for the fine-grained resource profiles with large inputs to the cyclic and homogeneous workflows, which we use as a representative examples (for other graphs please see our repository). We omit the costs for the cyclic workflow for these resource profiles, because only ChunkFunc manages to fulfill all SLOs for large inputs. The SLO adherence of all but one workflow follows the pattern shown in Figure 7.7a, where ChunkFunc meets all SLOs, StepConf meets some, but closely misses most SLOs, and SLAM misses all SLOs. The exception is the homogeneous workflow, shown in Figure 7.7b, where ChunkFunc fulfills all SLOs, StepConf misses all SLOs, and SLAM fulfills a little less than a quarter of the SLOs.

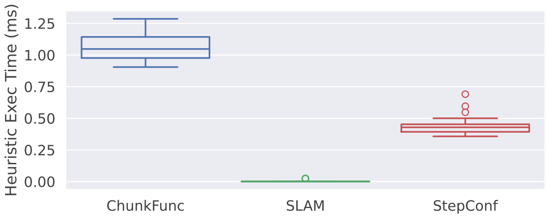


Figure 7.8: Cyclic WF Large Input Heuristic Execution Times for Fine-grained Resource Profiles.

ChunkFunc is the only heuristic that meets all SLOs for all input sizes for the BO-predicted profiles. Across all six synthetic workflows and three input sizes, StepConf meets 53% of the SLOs, with the lowest value being 45% for the homogeneous workflow and the highest being 59% for the random workflow. SLAM meets 65% of all SLOs, with 62% and 74% being the lowest and highest values respectively. For ChunkFunc this yields an increase in SLO adherence of 89% over StepConf and 54% over SLAM.

For the costs of the cyclic workflow, ChunkFunc amounts to only 49% of the costs of SLAM and 98% of the costs of StepConf. For the homogeneous workflow ChunkFunc requires 77% more costs than StepConf, but in any other case, ChunkFunc is cheaper. On average ChunkFunc reduces costs by 52% compared to SLAM. Compared to StepConf ChunkFunc is 5% more expensive overall, because of the homogeneous workflow. For the other five workflows, ChunkFunc is on average 9% cheaper than StepConf.

The cost difference between ChunkFunc and CF-Oracle, which knows all function outputs when computing a critical path, is negligible. Across all experiments with all workflows and resource profiles ChunkFunc is only 1% more expensive on average, which shows that its critical path estimation works well for keeping costs low, while fulfilling the SLOs.

Figure 7.8 examines the execution times of the three heuristics for the cyclic workflow. We log the execution time for computing each resource profile in a simulation and, then, compute the mean time for determining a single resource profile. We accumulate these values across all SLOs for an input size. Since SLAM only performs max-heap operations it is the fastest. ChunkFunc and StepConf both compute paths through a DAG and show a similar performance, with median values close to 1 ms and 0.5 ms respectively. Since ChunkFunc fulfills all SLOs, the slight increase in computation time over StepConf is justifiable and since it is marginal, it does not affect the user experience when invoking a workflow.

7.6.3 Takeaways

While automatic profiling causes some up front costs, workflows are typically executed for months or years in production. For example, profiling the `merge-videos` function in the VidPro workflow took 106 minutes, which on a GCP `c2-standard-30` VM with SSD amounts to a one time cost of about $2.27, which amortizes quickly since ChunkFunc may reduce function execution costs by up to 61%. New versions of a function can reuse existing performance profiles. Reprofiling is only necessary if the changes affect the function's performance. This can be revealed using a performance test in the continuous integration pipeline.

The number of inputs that should be profiled for a particular function to obtain the best resource optimization results depends on the function and its typical uses. A suitable approach for a production system is to monitor a function's live usage for a representative period, e.g., one week. A clustering of inputs can be used to identify the ideal number of inputs for profiling and to obtain sample input data as well. An automation of this step is currently out of scope, but should be considered as a future expansion.

State-of-the-art approaches for resource optimization do not consider input sizes, causing them to underestimate function response times, especially for large inputs. This leads to the selection of too weak resource profiles, often violating the SLO. ChunkFunc is the only heuristic that always meets the SLO because its input size-aware heuristic provides more accurate estimates for function response times. The analysis of all results shows that the more accurate critical path estimation and input data size awareness of ChunkFunc fulfills the SLOs in all test cases, an increase of a factor of 1.04 to 2.78, with respect to the state-of-the-art and a maximum cost saving of 61%. The advantage of input data size awareness becomes more apparent as the input data size-dependent response time of the functions increase, i.e., ChunkFunc performs better in processing intensive workflows, such as video encoding, where a badly chosen resource profile has a large effect.

In some cases the input data size is not the most decisive factor for function response time because other properties of the input are more important. For example, a video's file size is determined by its length and bitrate. However, when encoding a video, as we do in the VidPro workflow, the video's resolution has a much greater effect on the encoding duration than its bitrate. Another example is earth observation data from satellites: the image resolution and raw data size are always the same, but the processing complexity can change depending on whether the image shows the ocean or an urban area. To encompass such cases, ChunkFunc's input size parameter can be generalized to an abstraction that represents an arbitrary numeric property of the input, which affects processing time the most. In many cases this is the file size, but in some cases it may be another property. For example, for VidPro and FaceDet we use the product of $resolution \times length$ as the "input size." For satellite imagery a preprocessing function can be used to determine the complexity, which will be used as the "input size" for the next function.

Our Workflow Optimizer uses bucketing for selecting a performance profile for an input that does not exactly match one of the pre-computed performance profiles. Doing this instead of linear interpolation between profiles makes it easier to fulfill the SLOs. In the future a Gaussian Process could be bootstrapped with the pre-computed profiles and, then, used to infer the resource profile for such unknown inputs.

7.7 Summary

Fulfilling a serverless workflow's end-to-end response time SLO at minimal cost requires selecting appropriate resource profiles for all functions of the workflow. Many functions exhibit different performance characteristics for different inputs, necessitating the selection of different resource profiles based on the input.

We presented ChunkFunc, a framework for input data size-aware resource configuration in serverless workflows. We formulated an optimization problem to find function configurations that meet performance-based SLOs, while minimizing cost.

The ChunkFunc Profiler executes functions with their typical inputs to create input-size dependent performance models for them. Bayesian Optimization is used to guide the profiling, while building a surrogate model that approximates the performance profile of the function – this significantly reduces the number of configurations that need to be profiled. The Gaussian Process, which serves as the surrogate model, is used to infer the performance for configurations that are not explicitly profiled.

The ChunkFunc Workflow Optimizer leverages the performance profiles to adapt the configuration of functions in a workflow, based on their current input sizes, to meet performance-based SLOs while minimizing costs. We evaluated ChunkFunc against SLAM and StepConf and showed that it increases SLO adherence by a factor of 1.04 to 2.78, while reducing costs in many cases. This shows that input data size-aware resource configuration provides a significant advantage in serverless workflows with highly fluctuating input sizes.

CHAPTER 8

HyperDrive: Scheduling Serverless Functions in the Edge-Cloud-Space 3D Continuum

Scheduling serverless workflows in the Edge-Cloud continuum necessitates careful consideration of the network SLOs between individual functions to ensure adherence to the end-to-end response time SLO of the workflow. Extending the Edge-Cloud continuum with low earth orbit satellites to form a 3D continuum gives rise to new opportunities, such as global coverage with datacenter-like compute services and more efficient processing of earth observation data from monitoring satellites, but it also brings new challenges. HyperDrive is an SLO-aware serverless scheduler specifically designed to address challenges introduced by seamless execution of workflows across Cloud, Edge, and space, such as consideration of network QoS across the entire continuum and awareness of satellite temperature and satellite battery recharging capacity.

8.1 Introduction

As of 2024, there are over 8,000 low Earth orbit (LEO) satellites orbiting the Earth [172]. Satellites have traditionally communicated with each other via ground stations. Lately, inter-satellite links (ISLs) aim to connect satellites and create a large orbital network topology [21]. Starlink is currently the largest LEO mega-constellation with about

This chapter is based on the paper T. Pusztai, C. Marcelino, and S. Nastic, "HyperDrive: Scheduling Serverless Functions in the Edge-Cloud-Space 3D Continuum," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*, 2024.

7,000 satellites in orbit [216] and almost 12,000 total satellites approved by the FCC, which must be launched by 2028 [99]. By 2029 a second LEO mega-constellation is planned to be available with 3,236 satellites [76] and more competition is solicited by the FCC [216]. ISL capability allows LEO satellites to act as ground edge nodes, processing data directly in orbit and near the data source, such as Earth Observation (EO) satellite data. This opens up opportunities for new computing paradigms in space, such as Serverless Computing.

To address the environmental heterogeneity of the Edge-Cloud-Space Continuum, Serverless platforms need scheduling mechanisms that identify environmental properties and their current conditions to deploy functions and meet their requirements [166]. Most common scheduling approaches focus on meeting requirements based on resources, network, application and energy [24, 85].

Resource-Aware Schedulers [237] ensure that functions are executed on nodes capable of handling their computational requirements to prevent overloading any single node, which could lead to performance degradation or failures. In the Edge Cloud Space Continuum, resource-aware scheduling mechanisms [197, 17] dynamically allocate functions considering the infrastructure-specific resource characteristics such as CPU capacity and architecture, memory, and GPU. In Orbital Edge Computing (OEC), scheduling mechanisms [226, 30] address specific orbit characteristics such as satellite infrastructure resource and energy costs to transfer the data between satellites or to the ground stations. However, current approaches do not consider all aspects of Edge, Cloud, and Space as a unified continuum. They neglect the impact of resource temperature and heat generated by the task execution. Due to the substantial temperature variations on satellites between the daylight and eclipse periods of an orbit, tasks that require intense computation can produce too much heat, putting satellite components at risk of damage from overheating [138, 40].

Network-Aware Nodes at the edge typically have different network characteristics than cloud nodes. These network characteristics include variations in end-to-end latency, bandwidth availability, and link reliability [24]. Network-aware schedulers [197, 220] consider these characteristics to optimize function placement, ensuring efficient and reliable communication. In OEC, schedulers [226] typically also address the intermittent ISL communication between satellites and high latency communication with ground stations. However, existing OEC schedulers are not built for serverless functions, so they cannot guarantee the complete execution of serverless workflows across the Edge Cloud Space Continuum. Existing schedulers do not consider the positions of satellites, which is essential to ensure the seamless execution of serverless workflows from orbit to the Edge and Cloud. Therefore, existing schedulers fail to ensure that serverless functions can start, complete, and transfer all required data within the connectivity range of the satellite network.

Application and SLO-Aware Applications have SLOs that define the expected performance and availability during their execution. To meet these requirements, SLO-aware schedulers [186, 187] need to consider not only infrastructure properties such as resource availability, but also workload characteristics. Although OEC schedulers ensure functions can execute in a specific node, they do not guarantee workload requirements, i.e., SLOs, such as maximum latency.

Energy-Aware Schedulers consider the current power source and estimated task power consumption during the placement process. Energy-aware scheduling [123, 42] is crucial to prevent battery-powered devices from running out of power and to reduce overall power usage. By optimizing energy usage, schedulers ensure prolonged operational lifespans for edge devices and enhance sustainability, thus optimizing performance and longevity in the Edge Cloud Continuum. In OEC, energy-aware schedulers [226, 265] consider also the energy necessary to transmit the data either to other satellite nodes or to the ground station. However, existing schedulers overlook the satellite position during the energy consumption estimation. Despite tasks requiring a certain amount of power, the satellite can auto-recharge its batteries during the daylight periods.

Although current Serverless scheduling approaches address the heterogeneous devices on the Edge, they are not suitable for the specific environmental properties of the Edge Cloud Space 3D Continuum, such as satellite position and heat generation. Moreover, the current orbital scheduling approaches lack integration across the Edge Cloud and Space environment, essential for latency and function execution across the 3D Continuum.

In this chapter, we introduce *HyperDrive*, a novel Serverless platform that seamlessly integrates Edge, Cloud, and Space Computing, creating a 3D Continuum. HyperDrive is part of Polaris[1], a SIG of the Linux Foundation Centaurus project[2], a novel open-source platform for building unified and highly scalable public or private distributed Cloud and Edge systems, which is now expanding into the 3D Continuum. HyperDrive leverages the specific capabilities of each layer of the 3D Continuum, such as Edge proximity to the data and satellite proximity to Earth observation data, to enable optimized serverless function deployment and execution.

Our main contributions include:

1. The *architecture of the HyperDrive Serverless Platform*, which introduces novel components and mechanisms tailored to the unique characteristics of the 3D Continuum. HyperDrive enables functions to be seamlessly executed anywhere in the 3D Continuum, optimizing performance and reliability by ensuring that workflow SLOs are met.

2. The *HyperDrive scheduling model* is the foundation of our Serverless platform's scheduler, which is the main focus of this chapter. The HyperDrive scheduling

[1] https://polaris-slo-cloud.github.io
[2] https://www.centaurusinfra.io

model considers constraints such as resource capacity, application SLO requirements, satellite temperature, and network load to minimize the end-to-end Serverless workflow latency.

3. Our *Heuristic Scheduling Algorithms for the 3D Continuum* enable the realization of the HyperDrive scheduling model using a flexible MCDM approach. It first filters out nodes that are not capable of hosting a function and, then, scores the remaining nodes according to multiple criteria to find the best suited node for a function. Our prototype implementation is available as open-source[3]. HyperDrive achieves 71% lower E2E network latency than the next best baseline approach.

This chapter has eight sections. Section 8.2 presents the illustrative scenario and research challenges. Section 8.3 shows an overview of the HyperDrive Architecture for a Serverless Platform in the 3D Continuum. Section 8.4 describes the Serverless Workflow Model, HyperDrive scheduling optimization model, and heuristic scheduling algorithms for the 3D Continuum. Section 8.5 details our implementation approach and describes the design of our experiments. Section 8.6 discusses the results of the experiments, and Section 8.7 summarizes our work on HyperDrive.

8.2 Motivation

To further motivate our work we present an illustrative disaster response scenario and leverage it to derive research challenges.

8.2.1 Illustrative Scenario

Early detection of wildfires in remote areas is critical to mitigate their effects. Our scenario (Figure 8.1) involves using a combination of drones, LEO satellites, and ground-based Edge nodes that compose a serverless workflow for real-time wildfire detection, inspired by [129, 47, 146, 242]. The drones operate in high-risk wildfire areas, such as California during the summer, monitoring specific zones and capturing video and sensor data to watch for signs of wildfires. They send the data to the nearest Edge node using streaming frameworks or, when out of range, transmit it to LEO satellites acting as in-orbit Edge nodes. Once a fire is detected, LEO satellites incorporate satellite Earth Observation (EO) data for processing. Our serverless workflow processes the data close to the source to improve latency and reduce network overhead. In some situations, functions are executed directly on LEO satellites due to the data's proximity to EO data and the high latency associated with downloading data to the ground.

Figure 8.2 shows our Serverless workflow with four Serverless functions, partially executed on the Edge, partially executed in-orbit and partially executed in the Cloud. During the *Ingest* stage, real-time videos are transmitted to Edge nodes on the ground or in-orbit.

[3] https://github.com/polaris-slo-cloud/hyper-drive

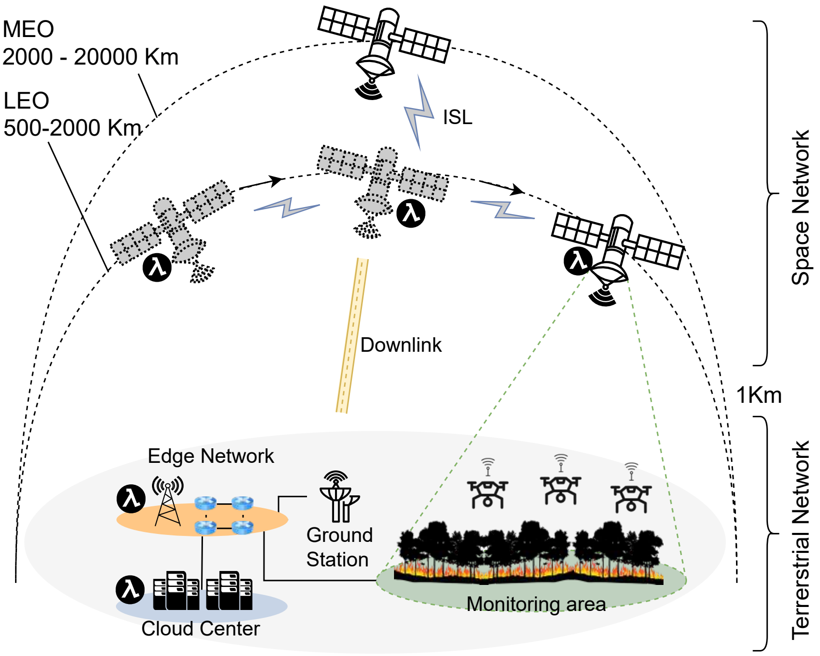


Figure 8.1: Illustrative Scenario: Wildfire Detection with On-ground and In-orbit Serverless Edge Computing

The *Extract Frames* function processes small video chunks received from Ingest stage and extracts image frames. *Object Detection* functions identify wildfire patterns in the extracted images, such as smoke, flames, or hotspots. The *Prepare Dataset* function prepares the data for resource-intensive tasks. The processed data is transmitted to the Cloud for storage and more resource-intensive tasks, such as machine learning model inference. In the Cloud, *Alarm trigger* functions evaluate the data and decide whether to trigger local emergency responses or deploy more drones to a specific area to confirm the wildfire before triggering an alarm.

Serverless computing allows dynamic scaling and processing close to the data source. By running Serverless functions directly on LEO satellites, we can combine data from the drones on the Earth and from EO satellites to process data as soon as they are produced. Atmospheric interference reduces link speeds to ground stations, typical speeds are around 300 Mbps[70]. Thus, downlinking data from EO satellites to Earth would take too long due to the large volume of data, e.g., each of the ESA Sentinel 2 satellites supplies high resolution images for a swath of 290 km in 13 spectral bands, producing about 1.5 TB of data per day [69, 3]. Since EO satellites only downlink to dedicated ground stations, the data may even be queued [243]. For Sentinel-2 “real-time” product availability is defined as “no later than 100 minutes after data sensing” [72], which violates the satellite data ingestion link SLO of the wildfire application. ISLs between EO satellites and LEO satellites are much better suited for large EO data volumes, since their speeds can be much higher – recently a 100 Gbps ISL from GEO to LEO has been demonstrated [71].

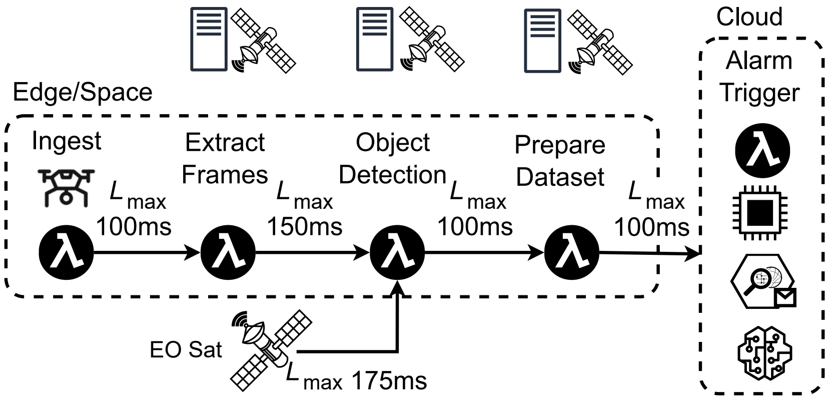


Figure 8.2: Simplified Serverless Workflow for Wildfire Detection

Hence, it is much faster to uplink a one GB ML model to the satellite than to downlink the EO data to a ground station. Drone videos are also moderate in size, e.g., a three minute 4K video from the FLAME2 dataset [39] amounts to 2.2 GB, which qualifies for uplinking to a LEO satellite in real-time.

Combining satellite EO data with drone data on LEO satellites allows reducing the time it takes to analyze and respond to wildfires. Additionally, it provides a reliable alternative when Edge nodes are out of range or experiencing connectivity issues. Scheduling the functions to execute in orbit ensures that wildfire detection and monitoring continue uninterrupted, even if ground-based infrastructure faces limitations. It allows immediate data processing and decision-making in orbit, reducing delays and ensuring continuous, real-time monitoring. As a result, we can decrease response times to wildfire threats. However, there are several challenges associated with scheduling Serverless functions in 3D continuum.

8.2.2 Research Challenges for Scheduling in the 3D Continuum

Based on the illustrative scenario, we identify several key requirements for scheduling serverless functions on LEO satellites in orbit as follows:

RC-1 *Satellite Availability*: Unlike Edge nodes, which have fixed positions, LEO satellites are constantly in motion as they orbit the Earth, which impacts their availability and communication windows [19]. A satellite must be within range of ① the drone, ② the EO satellite, and ③ the ground station to be considered available for scheduling. Specifically, the satellite needs to be within the drone's range to receive real-time video transmissions from Earth. At the same time, it must also be within the range of the EO satellite to receive and relay additional monitoring data. In addition, the satellite must be within range of ground stations, which have Cloud control planes for tasks such as scheduling. However, the term "in range" is more complex than direct line of sight. Since satellites can communicate via ISLs [96, 38], a satellite can be in range, if the bandwidth and latency via ISLs is acceptable for the purpose of the communication (e.g., data transfer). According to a recent study [156] Starlink's median roundtrip latency (client-LEO-Cloud) is 40-50 ms; the theoretical roundtrip latency between New York and London when routing exclusively through ISLs is 58-66 ms [96]. As satellites move in and out of range, the Serverless platform must continuously adapt, reallocating resources and re-establishing communication links. Therefore, satellite availability is more dynamic and complex compared to static Edge nodes.

RC-2 *Power Supply*: The scheduler must consider the satellite's power state, including its batteries' current charge level and the overall health of its energy storage system. Given the increasing computing power in satellites and the strict size constraints for some of them [138], the scheduler must be aware of the energy requirements of specific serverless functions to ensure that the satellite has enough power reserves to execute these functions without depleting its energy resources. Finally, a CubeSat's solar panels produce only up to 7 W of power [50], while batteries can have a density of up to 190 Wh/kg [167]. This means that a satellite might not be designed to fully recharge their batteries in a daylight period of an orbit. Thus, the scheduler must evaluate whether the power expenditure of its workloads can be compensated with solar power before the battery depletes.

RC-3 *Computing Capacity & Heat Generation*: LEO satellites are deployed with fixed and limited resources that cannot be patched or upgraded throughout their lifetime. These satellites are built to consume minimal energy and are equipped with minimal components to reduce weight and, consequently, launch costs. As computing increases, the temperature also rises. Since there is no atmosphere in space, heat dissipation mainly occurs through thermal radiation and lack of exposure to the sun. LEO satellites typically face temperatures from −120°C in the shade to +120°C when in the sunlight [78]. This situation can lead to prolonged high temperatures, affecting the performance of critical components such as the CPU [138, 248, 251, 47]. Therefore, the scheduler must consider not only the existing processing capacity but also the current temperature of the components and how long they potentially need to dissipate the heat.

RC-4 *Scalability*: Due to the fixed number of satellites in orbit and the increased costs associated with launching new ones, horizontal scaling presents a significant challenge. Compared to the ground data centers, where additional servers can be easily deployed to meet increasing demand, the satellite network is limited by the number of satellites currently in orbit. This physical resource constraint and fixed number of nodes make it challenging to auto-scale effectively to meet varying workload demands [180, 178].

RC-5 *SLO Awareness*: Serverless workflows must meet specific Service Level Objectives (SLOs) to ensure performance and reliability. These SLOs typically include minimal latency and bandwidth, which are essential for maintaining optimal service performance. Maintaining SLOs on the ground can already be challenging [183, 184] and these challenges are exacerbated by the network specifics, orbital movements, battery, and heat conditions of satellites [179]. Therefore, to ensure performance and reliability, the scheduler must consider the state of multiple nodes when enforcing workload SLOs.

RC-6 *Workflow Dependencies*: In a mixed environment, Serverless workflows can be executed on ground-based or LEO Edge nodes. The scheduler needs to take into account the workflow composition to identify the dependencies and interactions between the functions. Additionally, the scheduler must consider the placement of these functions within the workflow to ensure that interdependent tasks are located closely together to minimize latency and maximize efficiency [197]

8.3 Architecture Overview of a Serverless Platform for the 3D Continuum

HyperDrive is a novel serverless platform specifically designed for the 3D Continuum, as shown in Figure 8.3. To achieve that, our platform proposes six different layers: (a) an infrastructure layer that unifies the computing resources in the 3D Continuum, (b) a core platform layer for efficient and optimized function deployment and execution, (c) a function runtime layer for lightweight and low-latency execution, (d) a function model to allow developers change function behavior, (e) monitoring and tracing for

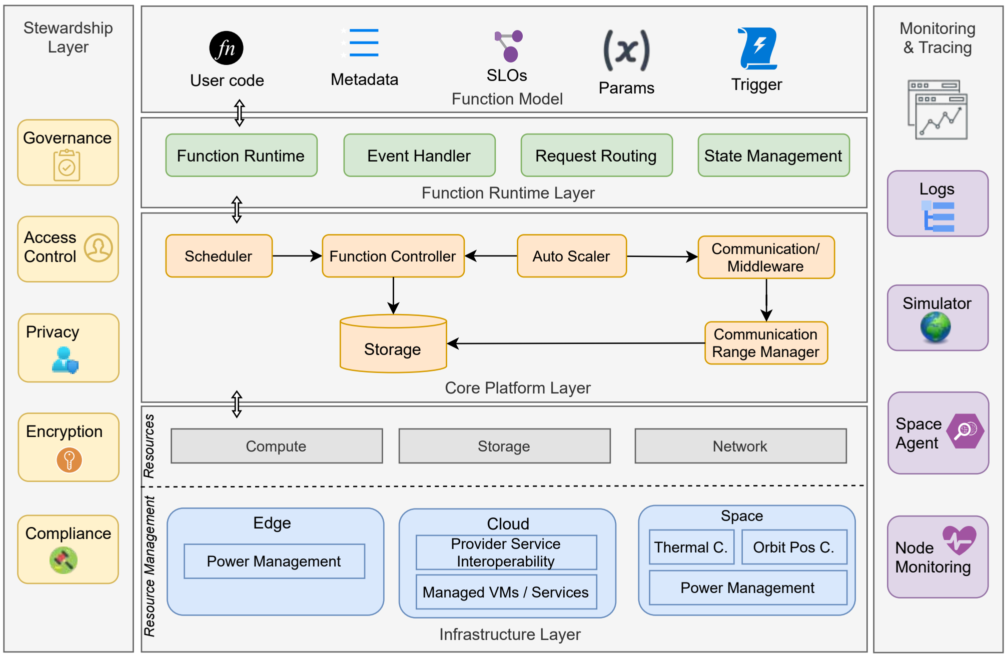


Figure 8.3: Architecture Overview of a Serverless Platform for the Edge-Cloud-Space 3D Continuum

real-time insights and (f) a stewardship layer layer composed of frameworks that enforce governance, security and compliance. Each platform layer introduces components to address the research challenges presented in Section 8.2.2.

8.3.1 Infrastructure Layer

This layer includes common computing resources across the Edge-Cloud-Space 3D Continuum, such as computing, storage, and network. Each computing layer, i.e., Edge, Cloud, and Space within the 3D Continuum, has specific properties that require tailored resource management. In the Edge layer, the HyperDrive infrastructure layer manages battery power to prevent Edge devices from running out of power. For example, by providing battery level information to the scheduler so that only drones with enough battery capacity execute the `Ingest` function in the wildfire serverless workflow. In the Cloud, it handles heterogeneous provider-managed services such as AWS S3 storage and Azure storage for storing high-resolution satellite images or more intense computing tasks such as running inference on machine learning models. In the space layer, the platform manages thermal regulation and power to prevent satellite depletion. Furthermore, the infrastructure layer provides satellite positioning information, which is critical for HyperDrive scheduler to place functions within range to ensure efficient data exchange between the functions. These computing resources create a unified infrastructure layer that adapts to the heterogeneous and dynamic requirements of the 3D Continuum, enabling the HyperDrive Serverless Platform to adjust to resources based on demand, ensuring seamless execution across the Edge-Cloud-Space Continuum. This is a key prerequisite to achieving our vision of self-provisioning infrastructures [162].

8.3.2 Core Platform Layer

This layer incorporates components responsible for managing and orchestrating tasks across the 3D Continuum. It manages the configuration, deployment, computation balancing [130], and auto-scaling of serverless functions, handling their lifecycle and scaling resources up and down based on the workload demand, such as the wildfire serverless workflow. Moreover, the storage enables HyperDrive to store function deployment properties and specific function configurations, such as parameters, state management settings, and SLOs. To allocate functions effectively, HyperDrive scheduler considers the 3D Continuum requirements described in Section 8.2.2. HyperDrive scheduler utilizes resource-based scheduling mechanisms, commonly used by various Edge and Cloud schedulers [197, 186, 187]. HyperDrive considers different requirements, including resource capacity, workload SLO, power supply, and satellite position, to make decisions using an MCDM approach. To ensure scalability in the large 3D Continuum, HyperDrive is a distributed scheduler that operates with multiple instances. Distributed scheduling requires keeping node state information in sync among the scheduler instances and handling scheduling conflicts. To address these two challenges, each HyperDrive scheduler instance obtains a function's candidate nodes and their states from the Monitoring Agent using sampling, similar to other distributed schedulers [54, 187], and handles conflicts using the MultiBind mechanism described in Section 8.4.3. By integrating Edge-Cloud-Space requirements, HyperDrive ensures the optimal placement and performance of Serverless functions within the 3D Continuum, thus meeting application demands and respecting boundaries between ground and space requirements such as latency and financial costs.

8.3.3 Function Runtime Layer

The Function Runtime layer consists of components such as Function Runtime, Event Handler, Request Routing, and State Management. The Function Runtime relies on lightweight frameworks such as WebAssembly to provide safety, isolation, and low-latency communication [143, 217]. In our illustrative scenario, the runtime utilizes function locality to reduce network overhead, ensuring satellites leverage local mechanisms such as inter-process communication (IPC) to exchange data between functions on the same host. Thus, the function runtime reduces latency and ensures that communication between co-located functions remains local, avoiding unnecessary ISL communication. Serverless stateless design pushes functions to leverage external services for state management [166, 194]. HyperDrive State Management leverages mechanisms such as short-term memory state [144, 91] to allow serverless workflows, like wildfire detection, to maintain their state between executions, thereby avoiding the overhead of external service communication. Due to the different properties, such as bandwidth, latency, and jitter, between Edge, Cloud, and Space, HyperDrive Request Routing optimizes load balancing by forwarding requests to functions in the vicinity, thus reducing latency by avoiding communication between functions cross-environment, such as Edge and space. The Event Handler manages events from different sources, such as image drones and EO data, to ensure proper function invocation. The components in this layer ensure a seamless execution of serverless functions to meet the workload requirements effectively. The function runtime layer offers lightweight mechanisms for executing functions on limited resource devices across the 3D Continuum.

8.3.4 Function Model

This layer introduces a function model that allows developers to define specific behaviors, such as SLOs and trigger types, in addition to the function code, parameters, and metadata. Developers can specify the type of event - such as streaming, asynchronous, or synchronous - that the function should process. In the 3D continuum, the function model enables users to react to specific satellite events, such as changes in orbit or satellite payload data received. Specifically, in the wildfire serverless workflow, drones at the Edge trigger `ExtractFrames` function using video streams, while `ObjectDetection` are triggered by single image frames as data input. Moreover, developers may specify certain SLOs, such as a maximum latency of 100 ms between two functions, for instance, between `ExtractFrames` and `ObjectDetection`. Without coding effort, developers can indicate whether functions are stateless or stateful. The HyperDrive Function Model layer abstracts the underlying infrastructure, enabling developers to manage serverless workflows without the complexity of coding or infrastructure management. Finally, this layer offers specifically tailored programming modes, e.g., to facilitate dealing with large-scale, heterogeneous data sources [212].

8.3.5 Monitoring & Tracing

This layer is composed of components that enable real-time tracking and monitoring such as Space Agent, Node Monitoring, distributed logging systems, and a simulator that enables developers to simulate functions execution without deploying the function on the expensive and limited infrastructure, e.g., on the satellites. The Monitoring Agent is designed to track and analyze key performance metrics across the 3D Continuum, including Edge, Cloud, and space infrastructure. It watches computing capacity, memory usage, and resource utilization across all nodes to prevent overloading and ensure efficient function execution. Additionally, it monitors network quality of service (QoS) parameters, including bandwidth and latency, to maintain compliance with workload SLOs. By monitoring the common properties of different layers, the Monitoring Agent enables seamless integration and reliability across the 3D Continuum.

The Space Agent is specifically designed to address the requirements of in-orbit computing. It is responsible for tracking the unique properties of the space environment, including the availability of LEO satellites, taking into account their rapid movement in orbit and their limited communication windows. Additionally, the Space Agent manages ISLs and ground-satellite network graphs to ensure that the satellite can meet the user-defined latency SLOs. It also monitors the satellite power supply, identifying the current charge levels of batteries and their position in relation to solar energy generation, to ensure that serverless functions are assigned only to satellites with sufficient battery capacity. Furthermore, the Space Agent monitors satellite thermal levels to prevent overheating caused by high computational load or prolonged usage, which could result in execution failures and potentially lead to long-term hardware damage. By addressing these space-specific requirements, the Space Agent plays a crucial role in optimizing the scheduling, deployment and execution of serverless functions across the 3D Continuum.

8.3.6 Stewardship Layer

This layer ensures serverless functions' secure, compliant, and efficient operation across the 3D Continuum. Its components enforce compliance with environmental and data protection regulations relevant to the workflow, such as wildfire monitoring. Encryption leverages mechanisms to protect stored sensitive information, while privacy mechanisms ensure that personal or location-based data is handled confidentially by the system. Moreover, Access Control implements role-based access and fine-grained permissions to restrict unauthorized access and actions. At the same time, the Governance component oversees these processes, enforcing policies and standards to maintain system integrity, security, and performance across the platform under expected conditions but also under uncertainty [163].

8.4 HyperDrive SLO-Aware Scheduler for 3D Continuum

The HyperDrive scheduler is designed to address the challenges that arise in the placement of serverless functions in the 3D Continuum first using an optimization problem and, then, using an MCDM approach. Without loss of generality, we assume that every serverless function is part of a serverless workflow, which we model as follows.

8.4.1 Serverless Workflow Model

A serverless workflow can be modeled as a DAG with every node representing an executable task, i.e., a serverless function or an operator, such as a condition, fork, or loop, and every link representing an invocation of the next node. The workflow DAG for our wildfire detection use case is is part of Figure 8.2; all executable tasks are by nodes with a λ sign. For the purpose of scheduling we refer to a serverless function instance as a *task*.

The workflow graph can be annotated with metadata relevant to its tasks. Each task node is annotated with information such as container image, resource requirements, preferred location, and SLOs. Since many network connections in the 3D Continuum are not as reliable as within a Cloud data center, tasks need to be able to specify special needs regarding the network QoS for incoming and outgoing links. To this end each workflow link can be annotated with network SLOs, specifically with maximum allowed latency, minimum bandwidth, maximum jitter, and maximum packet drop percentage.

In many cases serverless functions do not only depend on data from the predecessor function(s), but also on an external data source. In the 3D Continuum such an external data source may be, e.g., an S3 storage in a Cloud data center or high resolution data from an EO satellite. Workflow SLOs may result in special requirements for the connections to these data sources, i.e., network QoS SLOs. This entails that a workflow DAG must capture not only executable nodes, but also data source nodes and support SLOs on their outgoing links. The “EO Sat” at the bottom of Figure 8.2 represents an EO satellite node as a data source with its outgoing link providing EO data and imposing a max latency SLO of 175 ms to the `ObjectDetection` function. This metadata gives the HyperDrive scheduler all the required information to make a suitable placement of the workflow’s tasks.

8.4.2 HyperDrive Scheduling Model

Let a Serverless workflow be a DAG $\mathcal{W} = (\mathcal{F}, \mathcal{E})$, where each node in the DAG represents a function in the Set $\mathcal{F}$ and each edge in Set $\mathcal{E}$ represents the invocation of the next task. Let the network graph be $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is a Set of nodes and $\mathcal{L}$ the communication latency between the nodes. The scheduling goal to minimize the latency in the Serverless workflow $\mathcal{W}$ execution in the 3D Continuum, effectively mapping the workflow $\mathcal{W}$ onto the network graph $\mathcal{G}$. To achieve this, we consider the following constraints:

Resource Capacity: This constraint ensures that every node has enough resources to process the scheduled function, maintaining system stability and performance. Additionally, this constraint helps balance the system load across the nodes, optimizing the overall utilization of available resources. Therefore, the total resource demand D_i of function i on each node n in $\mathcal{N}$ must not exceed its availability resources R_n:

$$\sum_{i \in \mathcal{F}} D_i \leq R_n \quad \forall n \in \mathcal{N} \tag{8.1}$$

Network SLOs: This constraint ensures that data transfer between functions occurs within acceptable timeframes, ensuring that functions perform as expected. This means that communication between functions must meet performance criteria defined by the user to minimize delays. Thus, the SLOs latency S_{ij} must be met for each function invocation pair (i, j) in functions $\mathcal{F}$. The latency L_{nm} of the path between nodes n, m in $\mathcal{N}$ must not exceed the SLO S_ij:

$$L_{nm} \leq S_{ij} \quad \forall (i,j) \in \mathcal{F}, \forall (n,m) \in \mathcal{N} \tag{8.2}$$

Temperature: Managing thermal conditions not only protects the physical integrity of the nodes but also maintains optimal performance and longevity, specially in space where extreme temperature variations are common. Therefore, the temperature of each node n in $\mathcal{N}$ must not exceed its maximum allowed temperature $T_{\max}$, considering the maximum temperature caused by the satellite exposure to the sun and the temperature sum increase due to the execution of the each function T_{exc}:

$$T^n_{\text{orb}} + \sum_{i \in \mathcal{F}} T^{in}_{\text{exc}} \leq T^n_{\max} \quad \forall n \in \mathcal{N} \tag{8.3}$$

The scheduler goal is to minimize the total latency in the workflow execution by summing the latency L_{nm} between nodes n, m in $\mathcal{N}$ for each function invocation i, j in $\mathcal{E}$, where variables x_{in} and x_{jm} is a binary that indicates function placement to node. The optimization problem can be defined as follows:

$$\begin{aligned}
&\min_x \sum_{(i,j) \in \mathcal{E}} \sum_{n,m \in \mathcal{N}} L_{nm} x_{in} x_{jm} \\
\text{s.t.} \quad &\sum_{i \in \mathcal{F}} D_i \leq R_n \quad \forall n \in \mathcal{N} \\
&L_{nm} \leq S_{ij} \quad \forall (i,j) \in \mathcal{F}, \forall (n,m) \in \mathcal{N} \\
&T^n_{\text{orb}}(t_i) + \sum_{i \in \mathcal{F}} T^{in}_{\text{exc}} \leq T^n_{\max} \quad \forall n \in \mathcal{N} \\
&x \in \{0,1\} \quad \forall i \in \mathcal{F}, \forall n \in \mathcal{N}
\end{aligned} \tag{8.4}$$

The HyperDrive scheduling optimization model addresses key constraints of resource capacity, network SLOs, and temperature to guarantee efficient and reliable execution of Serverless workflows in the 3D Continuum. Minimizing total latency while adhering to these constraints enables the scheduler to make placement decisions across diverse environments, ensuring optimal performance and system stability. The consideration of satellite costs during scheduling is currently out of scope, since there are currently no pricing models for satellite nodes available.

8.4.3 Heuristic Scheduling Algorithms for the 3D Continuum

Given the high computational complexity of the aforementioned optimization problem, heuristics are needed to allow implementing the HyperDrive scheduling model for the 3D Continuum. We now examine the heuristic scheduling algorithms that approximate the aforementioned optimization problem. To this end we rely on an MCDM approach consisting of a sequence of filters that remove nodes that are not capable of hosting the task and scoring algorithms that determine the best suitable node among the eligible ones.

Vicinity Selection

Since the 3D Continuum may consist of tens of thousands of nodes, we need to perform a preselection of nodes before we can address the constraints of the optimization problem. To this end, HyperDrive contacts the orchestrator to select a set of candidate nodes that are located in the vicinity of the desired location specified by the task or in the vicinity of its predecessor task. The definition of the term "vicinity" can be configured independently for each part of the 3D Continuum. For example, for the Cloud any data center node within a radius of 500 km of the desired location may be selected, while the radius could be 200 km for Edge nodes, and 2,000 km for satellites. Akin to the vicinity, the total size of the candidates set and its composition can be configured as well, e.g., 500 total nodes consisting of 40% Cloud nodes, 40% Edge nodes, and 10% Space nodes.

Resource Checking

After selecting the set of candidate nodes, HyperDrive first filters out all nodes that do not meet the resource requirements of the task. Specifically, it checks the CPU architecture, CPU cores, memory, GPU (if present), local storage, and minimum battery charge (if the node has a battery) requested by the task.

Network SLOs Enforcement

HyperDrive uses a combination of filtering and scoring to ensure that the network QoS SLOs constraints for the incoming links of the task are fulfilled and the nodes with the best network properties are preferred. For filtering we use Algorithm 8.1. It iterates through all network SLOs for incoming links, originating from predecessor tasks and external data sources (if any) and queries the network QoS values for the lowest latency path between the candidate node and the node hosting the predecessor task or the data source. If the network SLO requirements are not met, the node is discarded.

For scoring we iterate through the aforementioned network paths again to determine the highest latency value We assign the highest score, i.e., 100, to the node with the lowest latency and zero to the node with the highest latency; all nodes in between are assigned proportional scores in the target interval.

Algorithm 8.1 Network SLOs Filter.

```
Input: $t$: Task to be scheduled;
$cn$: Candidate node;
$W = (V_W, E_W)$: Workflow DAG;
$N = (V_N, E_N)$: Network graph;
$S_t = \{(v, s) \forall v \in V_W \; s.t. \; (v, t) \in E_W \wedge s \neq \varnothing\}$: Network SLOs for incoming links of $t$;
Output: $true$ if $cn$ can host $t$, otherwise $false$;
for all $(v, s) \in S_t$ do
    $u \leftarrow$ GetHostNode($v, W, N$)
    $q \leftarrow$ QueryNetworkQoS($u, cn, N$)
    if Latency($q$) > MaxLatency($s$) then
        return $false$
    end if
    if Bandwidth($q$) < MinBandwidth($s$) then
        return $false$
    end if
    if Jitter($q$) > MaxJitter($s$) then
        return $false$
    end if
    if PacketDrop($q$) > MaxPacketDrop($s$) then
        return $false$
    end if
end for
return $true$
```

Temperature Optimization

The algorithm to enforce the temperature constraint is geared specifically towards the Space part of the 3D continuum to prevent satellites from overheating due to excessive workload when in the sunlight. Since a satellite that is close to overheating will reduce its computational power to prevent damage. Thus, HyperDrive aims to prefer satellites, where the new task will not cause a problematic temperature. This decision involves a complex estimate based on the current temperature of a satellite's compute unit, the expected duration of the task on the satellite's hardware, the required CPU and, possibly, GPU resources, the heat generated by these resources over the duration of the task, and the highest environmental temperature (based on in-orbit sunlight exposure) expected for the duration of the task. This is encapsulated in the scoring logic of Algorithm 8.2.

The algorithm first tries to get a duration estimate d_t for the task. This can be supplied by the user or through preceding profiling (on hardware similar to the satellite's) or the maximum response time SLO of the task can be used. If none of these values are

Algorithm 8.2 Temperature Optimization Scoring.

```
Input: $t$: Task to be scheduled;
$cpu_t$: CPU cores requested by $t$;
$gpu_t$: GPU cores requested by $t$;
$n$: Node to be scored;
$temp^n_{max}$: Maximum operating temperature for $n$;
$temp^n_{rec}$: Recommended high temperature for $n$;
Output: Score for the node $n$ in the range $[0; 100]$;
if NodeType($n$) $\neq$ "satellite" then
    return 100
end if
$d_t \leftarrow$ GetExpectedDuration($t$)
if $d_t == nil$ then
                        ▷ If $d_t$ is unknown use the current temperature to compute the score.
    $temp_{curr} \leftarrow$ GetCurrTemp($n$)
    return CalcScore($temp_{curr}, temp^n_{rec}, temp^n_{max}$)
end if

$temp_{inc} \leftarrow$ EstimateCompTempIncrease($n, d_t, cpu_t, gpu_t$)
$temp^{orb}_{max} \leftarrow$ EstimateMaxOrbitTemp($n, d_t$)
$temp^t_{max} \leftarrow temp^{orb}_{max} + temp_{inc}$
return CalcScore($temp^t_{max}, temp^n_{rec}, temp^n_{max}$)

function EstimateDuration($t$)
    $d_t \leftarrow$ GetExpectedDuration($t$)
    if $d_t \neq nil$ then
        return $d_t$
    end if
    return MaxResponseTimeSLO($t$)
end function
                        ▷ Estimates the temperature increase due to computation
function EstimateCompTempIncrease($n, d_t, cpu_t, gpu_t$)
    $temp_{inc} \leftarrow$ CpuTempIncrease($n, cpu_t, d_t$)
    $temp_{inc} \leftarrow temp_{inc} +$ GpuTempIncrease($n, gpu_t, d_t$)
    return $temp_{inc}$
end function

function CalcScore($temp_{exp}, temp_{rec}, temp_{max}$)
    if $temp_{exp} \leq temp_{rec}$ then
        return 100
    end if
    if $temp_{exp} > temp_{max}$ then
        return 0
    end if
    $range \leftarrow temp_{max} - temp_{rec}$
    $over_{rec} \leftarrow temp_{exp} - temp_{rec}$
    return $\lfloor (1 - \frac{over_{rec}}{range}) * 100 \rfloor$
end function
```

available the score is calculated based on the current temperature of the satellite. If d_t value is available, it is used in conjunction with the requested resources to estimate the computation-based temperature increase $temp_{inc}$. Subsequently, we determine the maximum expected environmental temperature $temp^{orb}_{max}$ during the orbit(s) within the duration of the task. The sum of these two temperatures is the maximum expected temperature for the satellite during the execution of the task and is used for computing the node's score. If the expected temperature is below the recommended temperature or above the maximum temperature, 100 or zero are returned respectively. Otherwise, a score is computed based on how much the temperature will go into the range between recommended and maximum temperature.

Multi Commit

Finally, all scores are accumulated for each node, and the nodes are sorted by their scores. The HyperDrive scheduler, then, contacts the orchestrator to assign the task to the highest scored available node using a multi-commit approach based Vela scheduler's MultiBind mechanism (see Section 6.3.2). Since multiple schedulers may be active, the orchestrator checks if the required resources are still available on the selected node. If that is the case, the task is committed to the node, a success message is returned to the and the scheduler updates the information in the DAG of the workflow instance. If the orchestrator reports that the required resources are no longer available, the result is a scheduling conflict, which most distributed schedulers resolve by rerunning the scheduling pipeline. To avoid doing this, HyperDrive tries committing the task to the second-best node and, if that fails too, to the third-best node, before triggering a rescheduling of the task. As shown in Section 6.4.3, the multi-commit technique decreases the number of scheduling conflicts by a factor of 10 with respect to immediately rescheduling the task.

8.5 Implementation & Experiments Design

To evaluate the HyperDrive scheduler we focus on the quality of the scheduling decisions and its scalability. Since HyperDrive is, to the best of our knowledge, the first serverless scheduler specifically designed for the 3D Continuum, we compare it against three theoretical scheduling approaches: Greedy First-fit, Round-robin and Random scheduling.

8.5.1 Implementation

The prototype of the HyperDrive scheduler is implemented in Python as available as open-source[4]. Since it is not feasible to run experiments on a low earth orbit (LEO) satellite mega constellation, we have connected our scheduler to a modified version of the StarryNet satellite constellation simulator [128]. The connection to the simulator is fully abstracted as an orchestrator interface, so that the simulator can be easily swapped. StarryNet normally executes Docker containers for all nodes. However, since we are interested in benchmarking the scheduling algorithms, we have replaced the containers with an in-memory nodes manager that tracks the available resources.

[4] https://github.com/polaris-slo-cloud/hyper-drive

We have implemented the 3D Continuum-specific scheduling heuristics described in Section 8.4.3. StarryNet precomputes latencies between adjacent nodes for the entire duration of an experiment. For each new time index, we use these latencies to update our network graph for network SLOs enforcement. Due to the absence of real satellite hardware information, we rely on reasonable estimates for the temperature optimizations.

8.5.2 Experiments Design

With our experiments we evaluate two critical aspects of the HyperDrive scheduler: i) scheduling quality with respect to latency and satellite temperature management and ii) scalability.

For assessing the scheduling quality we examine two major quality objectives. The primary objective is the latency achieved between the individual tasks of a serverless workflow and the E2E latency. The secondary objective is the intelligent selection of satellite nodes with respect to their temperature situation, i.e., satellites should be chosen, which will not overheat and reduce computational power while processing a task.

To set up the experiment we use TLE data, obtained on July 2, 2024 from CelesTrack[5], describing the orbits of 6,192 nodes of the Starlink[6] LEO satellite constellation. We deploy our wildfire detection use case, whose workflow is shown in Figure 8.2. We assign the `Ingest` function to a drone flying over a region of California, USA that is prone to wildfires and trigger the scheduling of the remaining functions as the simulation progresses. Since the StarryNet only supports satellite and ground station nodes, we model the drone as a ground station node. Since we evaluate the scheduling at the time when the second function needs to be placed, we do not require any movement from the drone, hence modeling it as a ground station does not limit our evaluation scenario. All experiments are run using Python 3.12 on Ubuntu 20.04 LTS on a Windows Subsystem for Linux 2 VM with 8 vCPUs and 8 GB of RAM. The VM is hosted on a laptop running Windows 10 22H2 on a Whiskey Lake-U generation Intel Core i7 processor.

We benchmark HyperDrive against the following theoretical schedulers, which we use as baselines:

- Greedy First-fit
- Round-robin
- Random selection

For evaluating the scalability we want to examine how HyperDrive scales with respect to the infrastructure size. To this end, we benchmark the placement of wildfire detection workflow on increasing infrastructure sizes. For Cloud and Edge nodes we simulate

[5] `https://celestrak.org/NORAD/elements/`

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

Table 8.1: Infrastructure Sizes used for Evaluation.

Satellites	Edge Nodes	Cloud Nodes	Total Nodes
1,008	100	10	**1,118**
2,016	200	20	**2,236**
3,024	300	30	**3,354**
4,032	400	40	**4,472**

nodes in the region the workflow is deployed in, while for satellites we simulate an entire constellation with the current 72 orbital planes of Starlink and an equal number of satellites per plane. Specifically, we use the infrastructure sizes described in Table 8.1 – node that the numbers in this table refer to our simulation only, which is limited by the resources of our host machine.

We execute five iterations of every scheduler's placement of the wildfire detection workflow on each of the four infrastructure sizes. We examine the achieved E2E latencies and temperature characteristics to evaluate the scheduling quality of all four schedulers and HyperDrive's processing time per task to assess its scalability.

8.6 Experimental Results

8.6.1 Scheduling Quality

To evaluate the scheduling quality we examine the network latencies achieved by the placements and the temperatures of the selected satellites (if any).

Figure 8.4 shows the mean network E2E latencies achieved by the four schedulers across all 20 experiment iterations, i.e., five iterations for each of the four infrastructure sizes. For clarity, the shown latencies are the sum of the network latencies only, without function execution times. The E2E network latency SLO, without function execution times, across all four functions of the wildfire workflow is 350 ms. While all schedulers, except for the Random scheduler, meet the E2E network latency SLO, HyperDrive clearly has the lowest latency, because it actively optimizes for it. HyperDrive's E2E latency is 71% lower than Round-robin's, which is the second best. While Greedy First-fit and

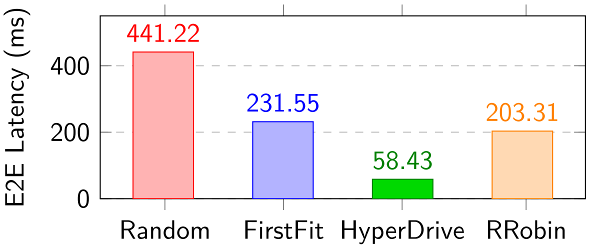


Figure 8.4: Wildfire Detection Workflow Mean E2E Latency per Scheduler.

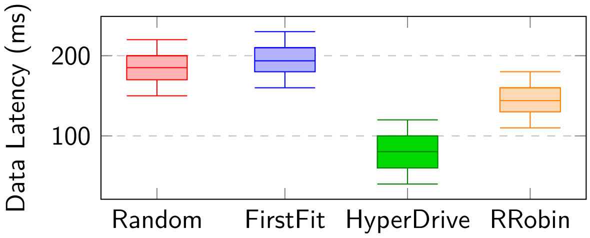


Figure 8.5: Data Latency per Scheduler

Round-robin meet the E2E network SLO, they violate individual function network SLOs in about 33% of the cases for Greedy First-fit and in 30% of the cases for Round-robin. HyperDrive fulfills all function network SLOs.

Apart from inter-function network SLOs, the wildfire detection workflow also defines a network SLO for an EO satellite data source. The `object-det` function requires a maximum latency of 175 ms to the respective EO satellite. Figure 8.5 shows the EO data latencies achieved by the schedulers. Random and Greedy First-fit violate the SLO. HyperDrive and Round-robin fulfill it on average, albeit Round-robin violates the SLO in 35% of the cases. HyperDrive always fulfills it, because its filtering does not allow scheduling on nodes that would violate the SLO.

The secondary optimization objective after the network latency, is satellite temperature measurement to avoid overheating. Figure 8.6 shows a heat map for the three scheduled functions that documents cases when the functions are scheduled on satellites and their temperature exceeds the recommended operating temperature. HyperDrive places 34 of the total 60 function instances (56.7%) across all iterations on satellites and never exceeds the recommended temperature. The Random scheduler places 56 of 60 function instances (93.3%) on satellites and exceeds the recommended temperature in 23 (41%) of these cases; in three cases it even exceeds the maximum operating temperature. Round-robin schedules all 60 function instances on satellites and exceeds the recommended temperature in one third of the cases; in four cases it exceeds the maximum operating temperature. It should be noted that as the number of Edge nodes increased in the two larger infrastructure sizes, HyperDrive selected more Edge nodes instead of satellites, due to their favorable network latencies; for the smaller two infrastructure sizes 86.7% of the nodes were satellites, while for the larger two only 33.3% were satellites.

8.6.2 Scalability

The goal of the scalability evaluation is to see how HyperDrive's performance evolves as the infrastructure size increases. Figure 8.7 shows the mean scheduling latency for each of the three serverless functions as well as the overall average. Since the prototype implementation is not connected to a real orchestrator and manually performs the vicinity selection with a linear search, we disregard the nominal scheduling latency values and focus on how they evolve with increasing infrastructure sizes.

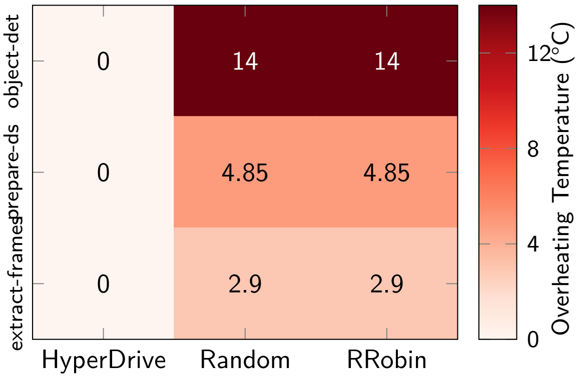


Figure 8.6: Scheduling Overheating Map.

It is evident that HyperDrive's performance scales linearly with the infrastructure size. The `object-det` function has a steeper incline than the others or the overall average, because needs to check twice as many network SLOs as the others, because it has a data source network SLO. Nevertheless, its increase remains linear.

8.6.3 Discussion

As previously seen, HyperDrive is the only scheduler specifically designed for the challenges of the 3D Continuum. HyperDrive excels at choosing between satellite and terrestrial nodes, depending on what benefits the network SLOs the most. As more nodes are available the quality of its scheduling decisions improves, e.g., the mean network latency between functions drops by 73% in the larger two infrastructures compared to the smaller two infrastructures.

Larger infrastructures yield better scheduling results, but they also increase processing time. Increased processing time, however, does not offset the benefits of optimized scheduling, because transfer and processing times of EO data are orders of magnitude greater than the scheduling duration. Additionally, scheduling on LEO satellites typically

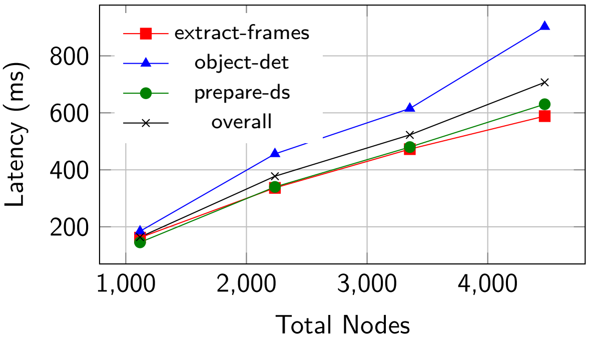


Figure 8.7: HyperDrive Scheduling Latency Across Infrastructure Sizes.

does not require high scheduling throughput, due to the type of applications that are expected to be deployed, e.g., federated learning in space or at the Edge [147, 83], advanced automotive use cases [262], monitoring applications [263], or disaster relief [242].

Finding multiple shortest paths through a large network graph is the biggest concern to the performance of HyperDrive. While HyperDrive scales linearly with the infrastructure size, the path finding time can be reduced by using a hypergraph to reduce the number of links and by computing paths between regions instead of single nodes. Additionally, the paths can be periodically precomputed and cached by the orchestrator. This will be addressed by our future work.

Currently we assume the absence of congestion on the network routes, but as satellite usage increases, this will be considered in future work. Additionally, a dense constellation can provide multiple routes [96] between two nodes and prioritization can be employed for disaster response applications.

We evaluated HyperDrive in simulations. However, its scheduling algorithms can be transitioned to a physical system. To this end they must be connected to a real-world orchestrator, which supplies metadata about real satellites (as well as Edge and Cloud nodes) and which can deploy functions on these nodes.

8.7 Summary

Extending the Edge-Cloud continuum with low earth orbit satellites into the Edge-Cloud-Space 3D Continuum provides new possibilities for running compute-intensive distributed workflows from almost anywhere on Earth. Especially use cases that involve data from remote regions and/or Earth observation data from satellites, such as wildfire detection, benefit significantly from this new compute continuum.

We presented HyperDrive, a novel Serverless platform that is specifically designed to enable a seamless execution of Serverless workflows across the 3D Continuum. We discussed the unique challenges of the 3D Continuum, such as the short communication windows of the fast moving LEO satellites, solar power supply, and the possibility of overheating while facing the sun.

The HyperDrive scheduler enables the optimized placement of Serverless functions in the 3D Continuum by considering network SLOs, workflow and data source dependencies, and thermal conditions of satellites. HyperDrive is, to the best of our knowledge, the first Serverless scheduler for the 3D Continuum. We evaluate it against three theoretical baseline schedulers by scheduling a wildfire disaster response workflow with strict network SLOs and EO satellite data dependencies. HyperDrive achieves 71% lower E2E network latency than the best baseline and shows linear performance scalability with the infrastructure size.

CHAPTER 9

Related Work

SLO enforcement, Edge-Cloud scheduling, and serverless resource optimization have been the focus of a lot of work on academia and industry in recent years. In this chapter we discuss important works related to our contributions.

9.1 Complex SLO Definition and Enforcement

In this section, we explore work related to SLO Script, presented in Chapter 2, and Polaris Middleware from Chapter 3, which builds on top of it.

All big commercial cloud providers support automated elasticity of some sort. However, the vast majority only provides simple SLOs that use a lower and upper bound or an average threshold for a metric that is directly measurable on the system. Some requirements can also be expressed in more high-level terms, e.g., AWS allows specifying the targeted availability of a service [6] or the durability of a DB in “nines” (e.g., “four nines” meaning 99.99% availability). Nevertheless, availability and durability are only simple SLOs that address a single elasticity dimension and “nines” cannot be considered a business metric. Custom metrics can be provided at some cloud providers through a query language. However, specifying the metrics query in the SLO configuration reduces maintainability, e.g., our PromQL query to calculate cost efficiency was complex and specific to the components we used and would, thus, be cumbersome to maintain. Furthermore, each provider, such as AWS [7], Azure [150], and Google Cloud [88], uses its own mechanism for configuring the autoscaler, i.e., it is often not possible to write one configuration that works for all providers, fostering vendor lock-in The supported elasticity strategies are mostly horizontal and vertical scaling, with some exceptions, such as an elasticity strategy for the AWS DynamoDB [8] that allows increasing read and write capacities independently.

For Kubernetes, there are multiple autoscalers available – the most prominent being Horizontal Pod Autoscaler (HPA) [168], Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA) [234]. HPA allows a workload to scale out/in based on CPU or memory usage or metrics provided by the Kubernetes metrics server[1]. The only supported comparison methods are direct value comparison or average value (or percentage). The use of complex custom metrics requires an adapter API server to provide the custom metrics. The kube-metrics-adapter[2] allows expressing such complex metrics queries in PromQL. However, this approach is difficult to maintain, especially for complicated metrics that require large queries. Vertical Pod Autoscaler (VPA) can be used to scale up/down, albeit currently not in conjunction with HPA[3]. It allows configuration of the vertical elasticity strategy, but not of the SLO – the decision when to scale is taken automatically based on the current resource usage. The limits defined for the pod are respected though. Cluster Autoscaler (CA) does not scale single workloads, but the entire cluster by adding and removing nodes as needed. Its SLO is the time that is allowed to pass after a pod can no longer be scheduled on the cluster due to a lack of resources until CA resizes the cluster[4]. HPA, VPA, and CA all tie their SLOs tightly to the elasticity strategies. There is research that improves on VPA [195] and CA [250]. However, they focus on improving the performance of the elasticity strategy and the final result, but not on the possibilities for defining SLOs or decoupling the elasticity strategy from the SLO.

Wang et al. [252] combine horizontal and vertical scaling to achieve an availability SLO and reduce costs. The SLO is however, limited to availability. It is achieved through horizontal scaling, while vertical scaling is utilized to reduce costs if the SLO is fulfilled. This approach provides a complex elasticity strategy, but it falls short of supporting complex SLOs and multiple decoupled elasticity strategies.

SLO-ML [66] is a language that allows service consumers to define SLOs in order for the language runtime to choose appropriate cloud services and SLAs for the deployment. While facilitating the initial deployment of a workload, it does not provide support for runtime elasticity.

OpenSLO[5] is a specification that allows the definition of and interaction with SLOs. However, it currently foresees the specification of metrics queries directly inside the definition/configuration of an SLO, which is hard to maintain for SLOs that depend on complex metrics and which should be reused many times.

Some frameworks are specifically designed for combining raw metrics into combined metrics, but they are normally not integrated with an SLO runtime. MELA [157] is

[1] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough

[2] https://github.com/zalando-incubator/kube-metrics-adapter

[3] https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#known-limitations

[4] https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-are-the-service-level-objectives-for-cluster-autoscaler

[5] https://github.com/OpenSLO/OpenSLO

designed for cloud elasticity monitoring and allows combining metrics using a metric composition language. StreamSight [84] is a language and framework for generating optimized queries for distributed processing of streaming analytics in edge computing, which may also be used to combine multiple metrics.

There are some languages that allow defining SLOs using custom metrics and triggering elasticity strategies. For example, SYBL [43] is a language and runtime that allow defining complex constraints, i.e., SLOs, on cloud applications and their components. It supports the definition of custom metrics and can be extended e.g., with additional elasticity strategies. However, its implementation is tightly coupled to OpenStack and it lacks a plugin system, which would allow extensions without recompiling the entire runtime. rSLA [227] is an SLA definition language with runtime facilities that allows the definition of SLOs using raw and custom metrics and supports triggering arbitrary actions, e.g., scaling, upon SLO violations. Both, SYBL and rSLA support SLOs, custom metrics, and decouple them from elasticity strategies. However, they do not pass parameters that result from the SLO evaluation to the elasticity strategies, thus discarding possibly important information and allowing only generic actions, instead of parametrized elasticity strategies.

Predictive approaches and AI-based systems have also been proposed to handle SLOs and scaling [124, 109, 158, 60]. For example, Dustdar et al. [60] propose the use of a Markov Blanket to model the complex characteristics of Edge-Cloud applications and to derive suitable actions to maintain certain properties in the computing continuum.

9.2 SLO-aware Scheduling of Microservices in the Edge-Cloud continuum

In this section, we discuss scheduling approaches for long-running microservices in the Edge-Cloud continuum, which are related to our Pogonip Scheduler from Chapter 4 and our Polaris Scheduler from Chapter 5.

Among the large selection of available container orchestrators [199], the default schedulers of those typically used in production, such as Kubernetes [235] and Docker SwarmKit [56], often rely on greedy multi-criteria decision making algorithms. Their default configurations tend to spread containers over the cluster, which works well in a Cloud environment, but has drawbacks in heterogeneous Edge clusters, where network QoS is not uniform. Nomad[6] is also used in production environments and specifically supports Edge clusters; its default scheduler [97] is also multi-criteria decision making-based, but it has no support for network QoS SLOs.

Two-level schedulers, such as Apache Mesos [100] or YARN [244] are also commonly used in production. They do not coordinate containers directly, but other schedulers or execution frameworks. The first level assigns cluster resources to the execution frameworks

[6] https://www.nomadproject.io

at the second level – each of these frameworks has its own scheduler. Mesos uses an offer-based technique, where the first level uses a fair sharing or a strict priorities approach to offer resources to the second level scheduler, e.g., an Apache Spark[7] scheduler or an Apache Storm[8] scheduler, which then operates within the assigned resources. YARN is a request based two level-scheduler; its first level scheduler receives job scheduling requests and passes them on to a second level scheduler, which then requests resources from the first level. While Mesos and YARN are not aware of Edge cluster properties, we will investigate their two-level approach in our future work for developing a distributed scheduling framework. YARN supports plugging in different schedulers at the first level. The Capacity Scheduler [228] is aimed at multi-tenant systems and ensures that each tenant gets a minimum resource capacity. The Fair Scheduler [229] seeks to achieve a fair distribution of resources across the application frameworks managed by the second level schedulers. It supports three policies: i) FIFO prioritizes based on the submit time of an application, ii) Fair aims for a fair distribution of memory across the application frameworks, and iii) Dominant Resource First is based on [86] and first determines the dominant resource for each application framework (the most used resource with respect to its available capacity) and its usage share, then it aims to equalize these dominant resource usage shares across all application frameworks. With respect to fairness, Pogonip and Polaris Scheduler rely on a FIFO approach, which is acceptable, because in both cases we focus on network SLOs within a single application.

While these Cloud solutions represent interesting approaches, the differences between Cloud and Edge resources prevent their direct adoption in an Edge environment. Different Internet connectivity and bandwidth, as well as resource distribution, call for strategies that explicitly take into account the presence of heterogeneous resources and non-negligible network delays.

Various techniques have been used for scheduling at the Edge as surveyed, e.g., in [24, 27]. The most common approaches include mathematical programming, heuristics, and genetic algorithm (GA).

Mathematical programming exploits methods from operational research to solve the application placement problem, e.g., [106, 159]. For example, to save energy consumption, Huang et al. [106] model the mapping of IoT services on Edge devices as a quadratic programming problem. Although simplified into a linear formulation, it may require prohibitive resolution time when the problem size grows. The main drawback of mathematical programming solutions is scalability. For Pogonip we also proposed an ILP, but the placement problem is well-known to be NP-hard, therefore, efficient heuristics are needed.

Many heuristics approaches build upon the previously mentioned Cloud-proven schedulers and extend them for the Edge and a limited degree of SLO-awareness. Fahs et al. [74] and Rossi et al. [200] present orchestration frameworks based on Kubernetes to determine

[7] https://spark.apache.org
[8] https://storm.apache.org

the number and location of replicas that are necessary to meet the application QoS requirements. Eidenbenz et al. [61] propose a latency-aware Fog layer architecture for industrial applications. Santos et al. [205] consider network bandwidth in addition to latency (specifically, round trip time) in their Kubernetes scheduler extension. While these approaches focus on Edge computing and introduce a notion of network SLO, they fail to cover all aspects of network QoS, e.g., bandwidth variance, latency variance, and packet drop are missing. However, especially variances contain important information about the stability of a network connection and should be considered during scheduling.

Cérin et al. [33] propose a scheduling strategy for Docker Swarm that allows users to select one of three economically oriented SLA levels for their workloads to define priorities for the scheduler. While this strategy may yield economic benefits, it is not specifically designed for the Edge. Menouer et al. [148] present MCDM strategies that improve on the original Docker SwarmKit strategy, but also remain focused on the Cloud.

Aral et al. [14], like Pogonip and Polaris Scheduler, rely on a graph of the network to compute scores for the latency and bandwidth between user groups and the deployed services. However, they do not consider a microservice-based application as a whole, because they focus on the connection between the users and the service accessed by them. Faticanti et al. [75] model an application as a DAG, partition it between the Cloud and Fog, and compute a placement considering the throughput required between the nodes of of the application graph, other important network QoS parameters are largely neglected. Pallewatta et al. [175] consider fog nodes organized as a tree. When a fog node receives an application execution request, it uses a greedy heuristic to allocate microservice starting from the leaf nodes; if no resources are available, it can forward the request towards the parent node. This decentralized approach promises to reduce latency and network usage.

To solve the Edge placement problem, different works (e.g., [219, 253, 94, 267]) rely on genetic algorithms (GAs). For example, [267] introduces the concept of Edge sites to decentralize the resolution of the microservices placement problem optimizing the application response time. Each site uses a GA to decide which microservices and how many instances of them to place in the current site as well as those to propagate to the neighbor sites. Even though GAs considerably reduce the average time to find a good solution, especially for a large solution space, they may react slowly to changes of an Edge computing environment.

All previously mentioned policies, as well as our Polaris Scheduler, consider the problem of scheduling synchronous applications. However, in certain scenarios an asynchronous application may prove useful [221], where a (logically centralized) message queue supports the microservice communication. Although asynchronous applications are starting to be investigated in the context of load balancing [170], to the best of our knowledge, they are so far poorly explored in the context of service placement. Our Pogonip Scheduler is designed to fill this gap and place asynchronous microservice-based applications in the Edge-Cloud continuum.

While Polaris Scheduler and Pogonip are similar in their ways of treating network SLOs, they are complimentary in terms of their intended uses. Polaris Scheduler is designed to find a network SLO-compliant placement of a microservice-based application with a synchronous communication pattern that results in a complicated graph of dependencies among its components. The goal of Polaris Scheduler is to not only find an initial placement that fulfills the network SLOs, but to select nodes that are likely to fulfill the SLOs for a long time. Pogonip is designed to schedule applications with an asynchronous communication pattern that relies on a message broker. While the application graph is simpler here, compliance to network SLOs is equally important. Pogonip prefers placing microservices on Edge nodes and considers Cloud nodes for offloading, because its cost model assumes that the latter incur additional cost, while Polaris Scheduler treats Cloud and Edge nodes equally.

9.3 Distributed Scheduling

This section discusses distributed scheduling approaches and compares them to our Vela Scheduler from Chapter 6.

A *monolithic scheduler* is by definition not distributed, but included here for comparison purposes. It operates as a single instance that sees, and possibly even manages, the entire cluster state, which facilitates its implementation and makes collisions impossible. Supporting multiple scheduling policies complicates its implementation, because all policies must be contained in the monolith. As the cluster size grows or if multiple clusters need to be managed, like in the Edge-Cloud continuum, maintaining the entire state within a single scheduler instance becomes very challenging or impossible. The default schedulers of Kubernetes [237] and DockerSwarm [56] suffer from the typical issues of monolithic schedulers that we have previously mentioned. There are many works that focus on Edge-related capabilities for monolithic schedulers, e.g., Rossi et al. [200] propose a latency-aware Kubernetes scheduler for geo-distributed environments and Santos et al. [205] add latency- and bandwidth-awareness to their Kubernetes scheduler extension. Hailiang et al. [267] use a genetic algorithm that aims to reduce the response time for microservice-based Edge applications, but the algorithm runs offline, which inherently prevents it from being scalable. In general, none of these works consider a distributed approach to ensure scalability for the Edge-Cloud continuum, hence they cannot be applied in a globally distributed context like Vela.

Mesos [100] and YARN [244] are two-level schedulers that are frequently used in production [13, 120]. Their top-level is monolithic and the second-level relies on partitioning. For Mesos all scheduling decisions have to pass through the top-level scheduler, which may result in a bottleneck, and YARN's top-level needs to capture the entire cluster state and assign fine-grained resources to the second level, which may be an issue if the entire cluster state gets too big to fit into memory. The Fair Scheduler [229] in YARN allows achieving a fair resource distribution among second-level schedulers and Capacity Scheduler [228] ensures that each tenant of a multi-tenant system gets a minimum share of resources, but both approaches are designed for the Cloud, not the Edge. Epsilon [115] and OneEdge [208] are also two-level schedulers, whose first levels are monolithic. Epsilon's second level utilizes the shared state concept and supports autoscaling of the second-level schedulers. OneEdge uses sharding for the second level schedulers and it supports enforcing and E2E latency SLO. The major issue with these approaches is the monolithic first level, which can hinder scalability – Vela Scheduler aims to avoid this using its fully distributed, sampling-based approach, which does not require scheduler instances to maintain any cluster state beyond the node samples that are retrieved independently for each job. The downside of sampling is that the ideal solution may not be part of the sample, an issue that Vela tries to mitigate using its 2-Smart Sampling mechanism (additionally, we plan further improvements on this using AI-based sampling in future work). Hydra [46] builds on top of YARN and greatly improves scalability by federating multiple two-level clusters across multiple data centers, however it is designed for the Cloud and does not focus on Edge clusters.

Apollo [22], Omega [210], Tarcil [54], and ParSync [77] are shared state schedulers. Apollo's shared state is centralized and treated as read-only for the schedulers; the state can only be updated by status updates received from the cluster nodes. Omega supports different types of transactions to reduce scheduling conflicts. ParSync partitions the state internally and the scheduler instances get updates on different partitions on every synchronization iteration. The schedulers prefer to pick nodes from recently updated partitions to avoid relying on stale state data and, thus, reduce the chance for scheduling conflicts. Tarcil improves speed by sampling nodes from a shared state, but if the cluster is heavily loaded the sample size becomes very large, e.g., 82% of the nodes in one of their examples. Arktos [105] improves on the scalability of the Kubernetes scheduler by turning it into a shared state scheduler. All shared state schedulers suffer from the issue that the entire cluster state may become too large to be handled by a single scheduler instance and from the occurrence of scheduling conflicts. Our approach avoids the former issue by being fully distributed and drastically reduces conflicts using the MultiBind mechanism.

Sparrow [173] is a distributed scheduler designed for batch jobs that relies on sampling to collect nodes. The nodes are contacted directly, which is not feasible with globally distributed nodes. A late-binding mechanism is used to ensure that a job starts as quickly as possible: a job is assigned to the queues of all eligible nodes and the first node that dequeues the job gets to execute it. Sparrow cannot have scheduling conflicts, because jobs can always be queued on a node, an assumption that is only valid for batch processing systems. While Sparrow supports constraints for its sampling phase, they are evaluated in a centralized fashion. Vela Scheduler avoids contacting nodes directly to allow for global distribution and it specifically addresses scheduling conflicts, because it is not restricted to batch jobs.

Mercury [119] and Hawk [53] are hybrid schedulers that combine a monolithic scheduler for one type of jobs with a distributed scheduler for other jobs. Mercury divides the two scheduling approaches between “guaranteed” and “queueable” jobs, while Hawk divides them between “long“ and “short“ jobs respectively. Mercury solves conflicts by terminating queueable jobs in favor of guaranteed jobs, while Hawk avoids conflicts by queuing. Naturally, the monolithic part can become a bottleneck and many systems have a single job type, so these approaches are not always applicable. Vela Scheduler does not have this bottleneck and, while being primarily designed for microservices, it can support any job type through appropriate plugins.

9.4 Resource Configuration Optimization for Serverless Functions

Numerous works aim at optimizing serverless functions and applications to meet their SLOs, while minimizing cost.

9.4.1 Resource Configuration Optimization

In Chapter 7 we presented ChunkFunc. Solutions most similar to it, which aim to optimize the resource configurations of functions can be divided into two categories: i) approaches that build a performance model offline using a-priori profiling and ii) approaches that build the performance model online using monitoring data. Each category can be further subdivided depending on whether it supports single functions or entire workflows and by the algorithm type used to determine function configuration(s).

Offline Performance Modeling using A-priori Profiling. A-priori profiling typically executes the serverless function or workflow using a representative input or set of inputs under different resource configurations to build a performance model in an offline fashion, which is used to tune the function configuration(s) for production execution.

Approaches for single functions use a wide variety of algorithms. AWS Lambda Power Tuning [29] executes profiling runs and graphs the response times and costs to let users manually pick a configuration. CPU-TAMS [44] relies on regression modeling to create a "vCPU-to-memory model" for a particular platform. Subsequently, a single profiling run for a function using the maximum resources configuration suffices to perform optimization. CherryPick [5], albeit originally developed for big data analytics jobs, uses BO to reduce the number of profiling runs needed to find a configuration that matches, e.g., a response time SLO. MAFF [270] uses linear, binary, or gradient descent search to find a suitable configuration. It supports an active mode (a-priori profiling) and a passive mode (using monitoring data only).

Optimizing a workflow to meet an SLO is much harder, because the performance of one function can affect the available resource choices for subsequent functions. Most approaches for serverless workflows use graph algorithms on the workflow's call graph, or on a graph derived form it, to find suitable configurations. StepConf [254] estimates function execution times using a piece-wise fitting model, based on results from an "offline", i.e., profiling, phase and a quantile regression model for data transmission delays between functions. These estimates are used in combination with a workflow graph in an NP-hard algorithm and in a heuristic to find function configurations that fulfill the SLO, while minimizing cost. Lin and Khazaei [135] augment a workflow graph with information, such as profiling results and probabilities of taking a certain edge after executing a function node. After transformations, such as removing cycles, they obtain a "probabilistic DAG", on which they run a Probability Refined Critical Path (PRCP) Algorithm that progressively refines the transition probabilities, while determining function configurations. Costless [65] assesses, in addition to resource configurations, the possibility to fuse multiple functions into a single function and whether to execute them in the Cloud or on the Edge. It utilizes a "cost graph", which contains paths through all possible function fusion options, with each edge weight containing the execution time and cost of running the succeeding function node.

Some a-priori profiling approaches do not use graph algorithms, such as SLAM [204], which places all functions with their lowest resource configurations in a max-heap ordered by response time. It pops off the top function from the heap, increases its resources, and checks if the workflow's SLO is fulfilled now. If not, it reinserts the function into the heap (if further resource increases are possible) and continues.

Contrary to ChunkFunc, these approaches use either a typical input data size for profiling or an aggregation of profiling results over multiple input data sizes, but they do not differentiate between different input data sizes. While CherryPick can detect a large gap between expected performance and actual performance, e.g., due to changed input data sizes, and trigger a reprofiling, the current performance profile does not support multiple input data sizes. StepConf's approach is similar to ChunkFunc, however, it relies on piece-wise fitting to determine function performance, while we use BO and its Gaussian Process. StepConf uses the number of requests to a function to determine inter-function and intra-function parallelism. Through intra-function parallelism the number of requests indirectly influences the resources available to a function instance, however, the input size or complexity of a request does not. In accordance with pure serverless principles, ChunkFunc assumes each function instance processes one request at at time. Thus, the number of requests are only relevant to the autoscaler of the serverless platform and do not influence the resource configuration. Instead for ChunkFunc, the input size or complexity of each request influences the resource configuration, which leads to superior results, as shown in our evaluation. The approach of fusing functions in addition to optimizing their resources, as done by Costless, can serve as a complimentary strategy for finding SLO-compliant resource configurations. However, it cannot replace the awareness of input data sizes. The repercussions of not being aware of different input data sizes are exemplified by our evaluation of SLAM, which fails to meet the SLOs for inputs that do not match the expected size. Additionally, SLAM and PRCP precompute all configurations before executing the workflow, SLAM using a max-heap and PRCP on a graph. This entails that they cannot adjust if some functions take longer than expected. ChunkFunc executes its heuristic directly before invoking each function, which allows it to leverage information about the current status of the workflow and react if a previous function was slower or faster than expected.

Some systems tackle the resource configuration problem specifically for ML workflows and rely on the request frequency to influence the optimization. AsyFunc [177] reduces memory usage of Deep Learning (DL) inference workflows by not loading the entire model into every function and tuning intra-function parallelism. It uses the number of requests per second to determine the number of CPU cores to be assigned to a function to achieve efficient memory usage. HarmonyBatch [36] reduces response time and costs for model inference operations. It batches infrequent requests of different applications with the same model together on the same function instance. The request frequency and application SLOs are used to determine the resources and the batching. λDNN [256] optimizes the resources and number of serverless functions used for training a Deep Neural Networks (DNN) model, based on the model parameters and a time SLO. It iterates over all possible memory profiles, similar to ChunkFunc. However, even though λDNN tunes the entire training workflow, all functions are the same and use the same resource configuration in the end, which simplifies the problem. λGrapher [104] computes the memory configuration of serverless functions for Graph Neural Network (GNN) serving as the sum of the memory required by the runtime, the graphs, and the embeddings; CPU configuration is determined using Bayesian Optimization to minimize costs. These systems, which work well for ML workflows, can leverage a-priori knowledge about the functions and/or assumptions that ChunkFunc cannot use, cause it is designed for generic serverless workflows. AsyFunc can decide to not not load the entire model into every function, which is not possible for a generic system like ChunkFunc. HarmonyBatch can batch requests that use the same model on the same function instance. While ChunkFunc could do this too if the same input data is used, it would depend on the function type if this would provide a benefit, e.g., a video encoding function will always encode the video even if it is the same function instance. Contrary to ChunkFunc λGrapher can leverage prior knowledge about the memory requirements of the GNNs. While ChunkFunc uses BO to reduce profiling time, λGrapher uses it to find the CPU configuration with the minimal costs. This is possible, because λGrapher can leverage more a-priori knowledge than ChunkFunc has available.

Online Performance Modeling. Approaches that do not use a-priori profiling typically use historical or live monitoring data to build performance models in an online fashion.

Solutions for single functions use various algorithms. AWS Compute Optimizer [9] analyzes function invocations, their duration, errors, and the number of throttled invocations and uses ML (exact technique is unspecified) to make recommendations for configurations, but does not optimize automatically. Sizeless [62] uses a multi-target regression model trained on a large dataset obtained from monitoring synthetically generated functions. This allows it to predict the execution time of a function with monitoring from a single memory configuration only. Aquatope [269] relies on BO to learn the most suitable configuration that fulfills an SLO more quickly and aims to reduce cold starts as well. FaasDeliver [259] applies a new resource configuration to a function after every execution until its model is complete. It uses a Tree-structured Parzen Estimator to reduce the number of configurations that need to be explored. Libra [260] harvests unused resources from function instances and assigns them to other instances that require more resources.

It uses the first input to profile the function and to bootstrap multiple ML models; subsequent monitoring data updates these models. For every invocation, Libra predicts the required resources and the execution time based on the input size and harvests or adds resources based on these predictions.

Systems for optimizing workflows also use very diverse approaches. Eismann et al. [63] use Mixture Density Networks and Monte-Carlo simulations to predict costs of serverless workflows, based on their input sizes, but they assume that the functions' resources are already assigned and do not propose a solution to optimize them. COSE [198] relies on Bayesian Optimization to pick the resource configuration to apply to the next function execution, while it is building its performance model. Once it has sufficient data, it computes configurations and placements (Cloud or Edge) by solving an ILP problem. Orion [139] optimizes resource profiles, function co-location, and cold starts. It models function response times as distributions (one for each observed resource profile) to account for variability and finds correlations between the latencies of functions in a workflow. FireFace [131] initially does not rely on monitoring data, but uses static code analysis to extract internal features to allow it to estimate execution time under various resource configurations using a prediction model. Adaptive Particle Swarm Optimization using Genetic Algorithm Operators is, then, used to find the function configurations that harmonize SLO satisfaction and cost minimization. The prediction model is regularly updated using monitoring data. Jolteon [266] uses monitoring data to build its models and formulates a chance constrained optimization problem, which is solved by a convex optimizer after converting it using Monte Carlo sampling. Astra [111], relies on graph algorithms to approximate the solution to an optimization problem for analytics workflows. Like FireFace, Astra also does not use monitoring data, but it determines function execution times with a formula that uses the input data size and the computation time on a "unit size object" for a given resource configuration.

Online performance modeling does not require profiling or configuration of typical inputs, because it monitors the running system. However, while the performance model is incomplete, the SLO will likely be violated. Statistical methods, such as those employed by COSE, FaasDeliver, and Sizeless reduce this time, but they cannot eliminate it. Except for Astra and Libra, none of these solutions account for different input data sizes. Libra reassigns unused resources, but it does not directly support SLOs and it is limited to tuning a single function. While ChunkFunc uses profiling results from different inputs, Astra needs to determine the computation time on a "unit size object". This may be hard to do complex functions and its approach is limited to analytics workflows. The Mixture Density Networks and Monte-Carlo employed by Eismann et al. [63] could be an alternative to BO for creating performance profiles in the ChunkFunc Profiler. However, their approach also requires a collection of function monitoring data to train its model. The required volume of monitoring data is not specified, but the authors state that micro-benchmarks can be used to generate the data. This suggests that the volume is likely more than what BO needs during ChunkFunc profiling. Orion models response times as distributions to account for variability, but, contrary to ChunkFunc, Orion ignores that some variability may come from different input data sizes. Similar to SLAM, the approaches employed by COSE, FireFace, Orion, and Astra precompute the set of resource configurations prior to executing the workflow. Thus, unlike ChunkFunc, they cannot react to unexpectedly slower or faster function executions.

9.4.2 Vertical Scaling Approaches

Vertical scaling can be seen as the counterpart to resource configuration optimization for serverless functions, which is typically used in (micro)service-based applications. Many solutions use machine learning on historical and/or live monitoring data to predict scaling targets for combined vertical and horizontal scaling [201, 108, 203]. Other techniques, such as control theory [152] are also used. Approaches that focus solely on vertical scaling also rely on a variety of techniques, such as reinforcement learning [26, 258], rule-based [4], fuzzy logic [257], or regression [174, 222]. Vertical microservice autoscalers try to predict a configuration to fulfill a demand consisting of many user requests, whereas serverless configuration tuning, such as ChunkFunc, is applied on a per-request basis. Thus, traditional vertical scaling offers less flexibility since it can update resources only at coarser grain. Since Libra [260] allows harvesting unused resources from serverless function instances and assigning them to instances in need, it can be seen as a vertical scaler for serverless, albeit without direct support for SLOs.

9.4.3 Scheduling & Miscellaneous

Proper placement/scheduling of serverless functions can also play an important role in meeting SLOs. Many systems rely on monolithic schedulers. Knative uses the default Kubernetes scheduler [237], which uses a greedy multi-criteria decision making approach to find suitable nodes for the pods, but it is not SLO-aware. FnSched [223] relies on a greedy algorithm to place function instances on as few nodes as possible to allow unused nodes to be turned off. Skippy [197] is a scheduler for data-intensive serverless applications at the Edge. FaaSRank [261] uses reinforcement learning to automatically learn scheduling policies to to optimize function completion time. Owl [238] allows overcommitting physical resources with multiple serverless functions to improve resource utilization, while carefully monitoring and preventing service degradation. Monolithic schedulers have limited capacity, which means that the high scheduling frequency in serverless systems necessitates at some point a distributed scheduler to keep up with the load. Hydra [46] uses a federation of 2-level schedulers to achieve up to 40K scheduling decisions per second. Hermod [118] supports a distributed mode and uses early binding and hybrid load balancing to reduce slowdown compared to vanilla OpenWhisk scheduling. AuctionWhisk [18] adopts a distributed scheduling approach based on an auctioning mechanism. YuanRong [37] is a complete serverless platform that is used in production. Its uses a highly-scalable multi-level hierarchical scheduler that reduces cross-node communication. Scheduling is orthogonal to the SLO-aware resource configuration provided by ChunkFunc. ChunkFunc and other resource optimization frameworks rely on schedulers to place new function instances on the most suitable nodes to deliver the required performance.

Cold starts are known to affect the response times of serverless functions [249]; mitigation of cold starts is another complimentary approach to resource tuning to ensure SLO adherence of serverless functions. Caching or keep-alive guided by probability distributions is a common strategy for cold start avoidance and employed, e.g., by FaasCache [80] and O-RDC [176]. Pre-warming, as done by StepConf [254], IceBreaker [202], or Orion [139], is a complimentary strategy that often uses workflow context information to predict which functions will be called next. Another approach is to reduce the function startup time with alternative runtimes. Catalyzer [57] restores checkpoints of previously running functions instead of starting completely new instances. The Firecracker [1] microVM relies on a lightweight Virtual Machine Monitor and a stripped down Linux kernel that boots in 125 ms. WebAssembly runtimes allow multiple functions to be hosted in the same container and, hence, allow for faster startup than a container or VM [82].

Some works are dedicated to a detailed study of serverless functions and platforms, such as [214], which deeply analyzes compute and memory performance, scheduling, and the overhead of containers. Jindal et al. [114] use profiling, statistical methods, and Deep Neural Network methods to estimate how many concurrent invocations a function can support without violating an SLO – such information can be intergrated into the profilers of resource configuration optimizers. Liu and Niu [136] examine the current pricing practices of serverless providers and formalize them into a model. As an alternative to the current static pricing, they propose a dynamic auction-based pricing model. If providers decide to adapt their pricing models, this orthogonal research can be used to update the current pricing models used by ChunkFunc and similar solutions.

9.5 Scheduling in the 3D Continuum

In this section, we discuss work related to the 3D continuum and scheduling within this continuum.

9.5.1 Edge Cloud Continuum & Orbital Edge Computing

Several research studies [164, 59, 241, 60, 166] have proposed a paradigm known as the Edge-Cloud continuum (ECC). This paradigm involves integrating computing resources in different layers, composed of Edge devices such as sensors and wearables that produce data processed by low-resource Edge nodes close that are close to the Edge devices and high-resource Cloud servers. ECC aims to enable seamless integration between all the layers, allowing for efficient task distribution and improved application performance. By utilizing the advantages of both Edge and Cloud resources, ECC enables heterogeneous environments to adjust to computational needs and connectivity conditions. Our Hyper-Drive architecture from Chapter 8 proposes to expand the ECC to orbit by seamlessly integrating satellites as Edge nodes, thus creating a 3D Edge-Cloud-Space Continuum.

Lately, the extensive effort to expand on-orbit capability has led to further research to explore the implementation of core networks in space, offering several benefits such as enhancing mobile coverage in remote areas, facilitating direct device-satellite connections, and satellite computing [268, 16, 255, 243].

LEO satellites, like terrestrial Edge nodes, have limited computing capacity and like Edge nodes, satellites can be near data sources, such as Earth observation satellites. Therefore, the increase in LEO satellites in orbit allows data to be processed directly in orbit, near the data source, enabling Orbital Edge Computing (OEC) [55, 20]. Research [110, 35, 34, 67, 81] enables federated learning by leveraging their distributed localized data processing capabilities, enhancing real-time data analysis and decision-making in space applications.

The Tiansuan [251, 248] constellation leverages a cloud-native design to enhance onboard services, resources, and the development and management of satellite equipment. Tiansuan's cloud-native approach provides advantages in application deployment, scalability, and cost-effectiveness compared to traditional satellite designs, allowing for seamless integration of computing and networking. Tisuan's platform is composed of six different layers: Physical, Virtual Resource, Operating System, Container Service, Collaborative Orchestration, and Function Application. MobileViT [137] proposes a three-layer architecture to enable Satellite Internet of Things for Smart Agriculture. The infrastructure layer is composed of IoT devices such as sensors, drones and satellites. The capacity layer contains computing communication and caching while the application layer represents the different use cases such as Smart Agriculture, Smart Grid and Smart Port.

9.5.2 Space-as-a-Service

Research identifies emerging services in space [125, 98] such as *Constellation-as-a-Service*, *Satellite-as-a-Service* and *Payload-as-a-Service*.

Constellation-as-a-Service Mission MP42 [161] by NanoAvionics and Satellogic [206] aims to offer a satellite constellation service to IoT/M2M operators. Constellation-as-a-service allows businesses to deploy and manage their satellite network without launching their own spacecraft. This business model promises customized services such as dedicated satellites, dedicated rocket launches, in-country operation centers, access to a global ground station network, and dedicated platforms, including a private Cloud for image cataloging, processing, and storage.

Satellite-as-a-Service It proposes a shared multi-tenant satellite concept [73, 121, 264]. The shared-access model allows multiple missions to be hosted on a single satellite, enabling users to share platform and payload capabilities. The Satellite as a Service model includes ground segment validation using continuous integration and hardware simulators to ensure the reliability and safety of user-uploaded software. This approach uses existing satellite infrastructure and modern software tools such as CI/CD to create a flexible and cost-effective platform for space technology development.

Payload-as-a-Service It emerged after a shift from analog to digital satellite payload, which enabled satellites to serve multiple clients. Digital payload made it possible to customize satellite computing for specific purposes such as machine learning and earth observation [87]. Consequently, it enabled an alternative to expensive space infrastructure - *Payload-as-a-Service*. In this business model, a commercial operator owns and manages the satellite system, providing data (payload) to customers on demand. The service providers manage the satellite bus, integration, launch, and operations. On the other hand, clients access the data and may even operate the payload by starting/stopping data collection and monitoring [98, 213, 102].

All of these approaches focus mostly on satellites only, with very little or no involvement of terrestrial compute nodes. HyperDrive proposes a unified computing continuum that spans seamlessly across Edge, Cloud, and Space nodes. As such HyperDrive goes further than the aforementioned concepts. But the HyperDrive scheduler can also complement the Constellation-as-a-Service and Satellite-as-a-Service, because both allow customers to run their own workloads on satellites and, thus, require a scheduling mechanism.

9.5.3 Satellite Edge Task Scheduling & Offloading

In [93], an efficient framework is proposed for offloading inference tasks by partitioning Deep Neural Network (DNN) models into multiple satellites, including one high Earth orbit (HEO) satellite and multiple LEO satellites. The approach divides inference tasks, with the task owner executing the initial portion of the DNN and offloading the remaining portion to other satellites. FedLEO [67] proposes a distributed scheduling mechanism for LEO satellite constellations to overcome bandwidth limitations and intermittent connectivity. FedLEO leverages Satellite Edge Computing (SEC) to improve training efficiency by adding horizontal communication pathways among satellites and optimally scheduling interactions with ground stations. Unlike HyperDrive, FedLEO and task offloading approach considers data processing only satellites SEC, thus excluding Edge nodes in the ground for task placement.

An Orbital Edge (OE) [30] platform leverages ISL for satellite processing, reducing latency and leveraging distributed computational capabilities. It offloads computing tasks from ground nodes to a single satellite and its neighboring satellites. While the OE platform relies on satellite-ground communication links, which may cause data transfer delays, HyperDrive addresses each computing layer's challenges separately to create a unified Edge Cloud and Space Continuum platform.

CHAPTER 10

Conclusion

After having discussed all our contributions in detail and presented related work, we conclude this thesis with a summary of the important aspects, revisit our research questions, and present avenues for future work.

10.1 Summary

In Chapter 1 we defined the problem we focused on, i.e., the definition and enforcement of Service Level Objectives in the Edge-Cloud continuum. We presented our research questions, which we will revisit in Section 10.2, and we introduced our multi-layer architecture for a reliable SLO-aware orchestrator for the Edge-Cloud continuum. We now summarize our contributions that fit into this architecture as shown in Figure 1.1.

In Chapters 2 and 3 we presented SLO Script and Polaris Middleware for the definition and enforcement of complex SLOs. SLO Script provides a set of abstractions to define and configure complex workload-specific SLOs and elasticity strategies using a type-safe language. This enables the creation of a variety of SLOs, which are tailored to specific workload types, as well as, more efficient configuration of these SLOs, because incompatibilities or configuration errors are revealed before applying the configuration to the cluster. SLO Script's strongly typed metrics API allows for efficient querying and aggregation of metrics, as well as the definition of high-level composed metrics, and the orchestrator-independent object model enables the reuse of SLOs on multiple orchestration platforms. Polaris Middleware builds upon SLO Script to provide a framework for creating orchestrator-independent SLO controllers, which periodically evaluate their assigned SLOs and trigger elasticity strategies in case of a violation. A provider-independent metrics collection and processing mechanism extends the abstractions provided by SLO Script to enable the creation of composed, reusable metrics. Finally, Polaris Middleware provides a CLI for easy bootstrapping of Polaris projects and generation of scaffolding code.

In Chapters 4 and 5 we addressed SLO-aware scheduling in the Edge-Cloud continuum. The Pogonip scheduler handles network QoS-aware placement of applications that consist of long-running microservices, which communicate with each other asynchronously through a message broker. We formulated an optimization problem and presented a heuristic to approximate a solution for use in a Kubernetes scheduler. The Polaris Scheduler extended the ideas of Pogonip to synchronous microservice-based applications, whose interactions from a complex dependency graph. We capture these dependencies in a service graph and model the network QoS state in a network topology graph. Polaris Scheduler consists of an SLO- and network topology-aware scheduling framework and scheduling plugins that ensure that the network SLOs of the application are fulfilled and that the placement is likely to maintain the network QoS for a prolonged period of time.

Vela Scheduler, which we presented in Chapter 6, is a distributed scheduler designed to scale to tens of thousands of compute nodes dispersed over multiple clusters. Vela's architecture consists of two major components: an orchestrator-independent scheduler, which can be run in an arbitrary number of instances, and a cluster agent that is deployed in every target cluster and which acts as the connection to the local orchestrator. The scheduling pipeline consists of three stages: sampling, decision, and commit. The sampling stage leverages a two-level informed sampling mechanism that delegates sampling to the cluster agents to ensure scalability and which uses the requirements of the job to ensure that only nodes that are likely to be suitable for the job are returned to the scheduler. The decision stage filters and scores the nodes and picks the best three nodes as commit candidates. The commit stage first attempts to commit the job to the highest scored node and falls back to the second or third best node in case the resources have already been claimed by another scheduler instance. This approach reduces scheduling conflicts and the need for rerunning the entire pipeline.

In Chapter 7 we introduced ChunkFunc, an SLO- and input-aware resource optimizer for serverless workflows. Many serverless functions require different amounts of computational effort depending on the size of their inputs. ChunkFunc leverages this fact to assign different resource profiles to functions depending on their current input with the goal of minimizing costs, while adhering to the workflow's end-to-end response time SLO. The ChunkFunc Profiler automatically profiles each serverless function with typical inputs and uses Bayesian Optimization to reduce the number of profiling runs and to infer response times for not profiled configurations. The ChunkFunc Workflow Optimizer leverages these performance profiles to optimize the resources of each function in the workflow to meet the end-to-end response time SLO of the workflow.

In Chapter 8 we extended the Edge-Cloud continuum with low earth orbit satellites to form an Edge-Cloud-Space 3D continuum and we introduced the HyperDrive scheduler to place serverless functions in this compute continuum. HyperDrive includes a vision for a serverless platform for the 3D continuum, which enables seamless execution of serverless workloads in this continuum, e.g., to improve the responsiveness of disaster response and Earth observation applications. When scheduling a function, the HyperDrive scheduler considers not only the network QoS state such that the end-to-end response time SLO of the workflow is met, but also satellite-specific properties, such as satellite temperatures and possibilities to recharge batteries during the daylight periods of an orbit.

Finally, in Chapter 9 we discussed existing literature related to our contributions.

10.2 Revisiting the Research Questions

We now revisit the research questions that we presented in Section 1.2.1 and discuss how our work has addressed them.

RQ1. *How can complex Cloud-native SLOs be effectively monitored and enforced for long-running workloads in containerized infrastructures at application runtime?*

We have addressed this research question by Chapters 2 and 3. SLO Script enables the definition and configuration of complex workload-specific SLOs and elasticity strategies, which is a prerequisite to enforcing SLOs. One of the main design goals is to decouple SLOs from elasticity strategies. The strongly typed metrics query abstractions introduced by SLO Script and complemented by the processing mechanisms of Polaris Middleware facilitate the monitoring of long-running workloads, because they allow efficient querying and aggregation of metrics at runtime. These aggregations can be exposed as composed metrics, which enables decoupling of the metrics computation from metrics use in SLOs. Furthermore, it allows reusing metrics in multiple SLOs. The orchestrator-independent object model provided by SLO Script and the runtime mechanisms added by Polaris Middleware enable the realization of controllers that monitor complex SLOs using the previously mentioned metrics abstractions. The intrinsic separation of SLOs from elasticity strategies in Polaris allow users to leverage the same elasticity strategy for multiple SLOs, as long as the inputs/outputs are compatible. This significantly increases the number of SLO and elasticity strategy combinations available to the user without the need to duplicate code in additional controllers.

RQ2. *How can network QoS SLOs be leveraged to improve the scheduling of long-running workloads in the Edge-Cloud continuum and how can a scheduler scale with the growing infrastructure size?*

The first part of this research question is addressed by Chapters 4 and 5. The Pogonip Scheduler deals with network QoS-aware scheduling of Edge-Cloud applications, composed of long-running microservices that rely on a message broker for communication. These asynchronous applications exhibit a simplified communication pattern, because each component "only" communicates with the message broker. Hence, its proper placement is imperative for an SLO-compliant placement of the entire application. Polaris Scheduler extends the communication model to synchronous applications, which may exhibit complex dependencies among their microservices due to their direct calling behavior. We model these dependencies and their respective SLOs in a service graph, which we traverse in a breadth-first search to queue the application's microservices for scheduling. The extensible Polaris Scheduler framework relies on a greedy multi-criteria decision making approach to pick suitable nodes, while maintaining an acceptable scheduling throughput.

Each microservice is placed on a node that meets the network SLOs to the nodes that host the calling services and whose network QoS parameters exhibit a low variability, i.e., a high stability to maintain SLO-compliance for a long time. Meeting the network SLOs of the downstream microservices is more challenging, because their host nodes are not yet known. Thus, we resort to a heuristic, which ensures that the current node meets the most stringent network SLOs of the next direct downstream services.

The second part of this research question is addressed by Vela Scheduler in Chapter 6. Since even the most efficient monolithic schedulers have infrastructure size limits, the only option to scale the infrastructure further is to rely on distributed scheduling. Vela's architecture allows an arbitrary number of scheduler instances to be executed, where jobs may be submitted to any of those instances. The schedulers are orchestrator-independent, the only orchestrator-specific component is the cluster agent, which must be running in each cluster and which connects to the local orchestrator. Vela uses sampling to obtain candidate nodes for every job, but unlike typical existing solutions, Vela leverages the requirements of a job to sample only nodes, which are likely capable of hosting the job. To increase scalability sampling is not performed by a scheduler instance, but it is delegated to a selection of cluster agents, which have faster access to local node information. After sampling is complete, Vela chooses the best three nodes as candidates for hosting the job. Vela's commit mechanism processes tries committing the job first to the best suited node, falling back to the other two if the resources were already assigned by another Vela instance. This significantly reduces the number of scheduling conflicts that would otherwise trigger a rerun of the entire scheduling pipeline.

RQ3. *How can end-to-end response time SLOs of a workflow be used to optimize resources and placement of short-running serverless functions in the Edge-Cloud continuum?*

We addressed the first part of this research question in Chapter 7 with the ChunkFunc resource optimizer for serverless workflows. Since serverless functions are short-lived, the best lever to tune them for SLO compliance is to adjust their resource configurations. Any resource optimization for a serverless function needs a performance profile that indicates how the function performs with a particular set of resources. The aim of ChunkFunc is to provide precise optimization by considering the input size of a function when making a resource assignment. This necessitates input-specific performance profiles. ChunkFunc relies on profiling a function with typical inputs to obtain these profiles. However, to reduce the number of profiling runs, we leverage Bayesian Optimization to guide the optimization process. BO uses a Gaussian Process as a surrogate model to approximate the performance profile. Once we have collected enough profiling samples, we use the GP to infer missing parts of a performance profile. The ChunkFunc Workflow Optimizer computes the critical path from the current function to the end of the workflow using average function response times across all inputs and determines the relative contribution of the current function to that critical path to calculate a sub-SLO. Subsequently, a resource profile that fulfills this sub-SLO is chosen. This approach allows fulfilling end-to-end response time SLOs of workflows across all their typical input sizes, while minimizing costs.

Chapter 8 addressed the second part of this research question with the HyperDrive scheduler and platform. As an avenue to future work we extended the Edge-Cloud continuum with low earth orbit satellites to form the Edge-Cloud-Space 3D Continuum. LEO satellites exhibit very low latency and see an increase in computing power with every new generation, hence they can be treated similar to Edge nodes. Akin to scheduling at the Edge, network SLO adherence is of high importance in the 3D Continuum as well. HyperDrive uses a similar approach like Polaris Scheduler by using a graph to collect the requirements and call graph of a serverless workflow and relies on a network topology graph to compute shortest paths between nodes. However, scheduling on satellites introduces new challenges – apart from resources and network SLOs, HyperDrive also considers satellite temperature and the possibility to recharge the lost power using the solar panels when making a placement decision.

10.3 Limitations & Future Work

In this thesis we have proposed three research questions and discussed our contributions to addressing them. Like every technology, our work faces certain limitations and the research questions cover large fields. Thus, we now identify several directions for future work to complement and extend our work.

10.3.1 Complex SLO Enforcement for Long-lived Microservices

SLO Script and the Polaris Middleware enable the type-safe definition of SLOs, elasticity strategies, and controllers to enforce them. However, in our work we have only given examples of elasticity strategies that perform a single action, such as horizontal scaling or changing a microservice's configuration. Future work should investigate the possibility to flexibly combine multiple *elementary elasticity strategies* in *compound elasticity strategies*. For example, a compound elasticity strategy could combine the two elementary elasticity strategies of changing the configuration of a microservice and scaling it horizontally. The configuration change strategy is specific to the microservice, but the horizontal strategy can be the reusable one we already presented in this thesis. It is also worth investigating compound elasticity strategies that can be applied to entire application, which consists of multiple microservices, each of which is scaled with one or more distinct elementary elasticity strategies. Depending on the state of the SLO, the compound elasticity strategy decides which elementary elasticity strategies to trigger and for which microservices.

To reduce the number of SLO violations, the time when the elasticity strategy is triggered needs to be investigated. Our work has dealt with reactive triggering, i.e., the elasticity strategy is triggered once the SLO is violated. However, as noted in the related work, predictions and AI-based triggering are an active field of research. The combination of predicted metrics with existing Polaris SLO controllers is a promising avenue to explore, especially since our abstractions make it easy to replace an existing composed metric with a predicted metric.

10.3.2 SLO-aware Scheduling of Long-lived Microservices

Pogonip an Polaris Scheduler achieve good results for network SLO adherence and Vela Scheduler scales well to tens of thousands of nodes, but they all have limitations that should be investigated in future work.

The impact of mobile nodes, such as smart cars or drones, is currently only addressed in Pogonip and Polaris Scheduler by the assumption that the monitoring system regularly updates the network topology graph. However, this could lead to the deployment of a microservice on a node that will soon enter an area of poor network coverage, which may violate a network SLO. To mitigate this issue, movement predictions should be incorporated for mobile nodes, such that network QoS changes can be anticipated and included in the decision making process.

Scalability tests for Polaris Scheduler have shown that its shortest network path computation could cause scheduling throughput to decrease in very large clusters. To address this, we plan to investigate the use of hypergraphs for the Cluster Topology Graph to improve performance in such large cluster scenarios.

Vela Scheduler has a scalable design, but provides interesting avenues for future work in the fields of SLO support and further improvements to the sampling process. Extending Vela with support for network SLOs is an important, but challenging task, because of the large number of nodes. While this endeavor will profit from the above mentioned work on hypergraphs, the problem is exacerbated by the fact that a Vela scheduler instance has no knowledge of where other microservices of an application were scheduled. Yet, this information is needed for enforcing network SLOs. It is also needed for affinity/anti-affinity support. One possibility to enable such features would be to tie all microservices of a specific application to a particular scheduler instance. Vela's sampling process could be improved with AI-based sampling strategies that leverage information on the previous execution of similar jobs to produce even better samples. Additionally we want to further improve the scalability of our approach by increasing sampling performance and introducing sharding into the Cluster Agents.

10.3.3 SLO Adherence of Serverless Workflows

ChunkFunc has surpassed the state-of-the-art in SLO adherence, but it has some shortcomings that offer good possibilities for future work. We intend to adapt ChunkFunc for cost-based SLOs, such that it can find resource configurations that will yield the fastest possible response time for a maximum or average cost. The need to define typical inputs is a shortcoming, which may be addressed by adding a learning mode to automatically discover the most typical inputs, which could later be transferred to profiling. If the Workflow Optimizer discovers that an input size is not part of a performance profile, it currently uses the next larger input size that has been profiled. An interesting avenue is to investigate the use of a Gaussian Process to infer more precise resource profiles for such input sizes that are not part of the performance profile. In the longer term, our goal is to design a lightweight and extensible serverless-native framework for the development of serverless applications, which supports the definition and enforcement of SLOs, cold-start optimizations, and optimizations for inter-function communication.

For the 3D Continuum we plan to continue our realization of the HyperDrive Serverless platform. Since testing algorithms for the 3D Continuum requires simulation of thousands of satellites and terrestrial nodes, we plan to create a flexible 3D Continuum simulator. This simulator should be able to operate in a lightweight mode, which only computes node positions and the network topology, on a single machine for testing scheduling algorithms and in a full-fledged mode across multiple machines, where it allows executing the deployed workloads in containers or VMs. An important goal for the HyperDrive platform itself is to further to improve the coordination of function execution and satellite orbits, by placing functions on satellites that will be in the ideal position for a low-latency handoff to the next node when the function completes. We also intend to reduce scheduling complexity for large infrastructure sizes by using hypergraphs for inter-node path computations, based on the future work for Polaris Scheduler and Vela. This could be complemented by running shortest path computations regularly and caching the results for a certain time. Finally, we envision space-specific plugin components for our above mentioned lightweight serverless framework to enable the development the next generation 3D Continuum applications.

Overview of Generative AI Tools Used

No generative AI tools were used for writing this thesis and the publications it is based on.

List of Publications

This is an exhaustive list of all publications that this thesis is based on, as well as a list of the contributions of each author according to the CRediT taxonomy.

1. T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "SLO Script: A Novel Language for Implementing Complex Cloud-Native Elasticity-Driven SLOs," in *2021 IEEE International Conference on Web Services (ICWS)*.

   - T. Pusztai: Conceptualization, investigation, methodology, software, visualization, writing – original draft, and writing – review & editing.
   - A. Morichetta: Conceptualization and validation.
   - V. C. Pujol: Conceptualization and validation.
   - S. Dustdar: Supervision, conceptualization, funding acquisition, and writing – review & editing.
   - S. Nastic: Conceptualization, methodology, validation, and writing – review & editing.
   - X. Ding: Validation.
   - D. Vij: Validation.
   - Y. Xiong: Validation.

2. T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, "A Novel Middleware for Efficiently Implementing Complex Cloud-Native SLOs," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

   - T. Pusztai: Conceptualization, investigation, methodology, software, visualization, writing – original draft, and writing – review & editing.
   - A. Morichetta: Conceptualization and validation.
   - V. C. Pujol: Conceptualization and validation.
   - S. Dustdar: Supervision, conceptualization, and funding acquisition.
   - S. Nastic: Conceptualization, methodology, validation, and writing – review & editing.
   - X. Ding: Validation.
   - D. Vij: Validation.

- Y. Xiong: Validation.

3. T. Pusztai, F. Rossi, and S. Dustdar, "Pogonip: Scheduling Asynchronous Applications on the Edge," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

    - T. Pusztai: Conceptualization, investigation, methodology, software, visualization, writing – original draft, and writing – review & editing.
    - F. Rossi: Conceptualization, investigation, methodology, software, writing – original draft, and writing – review & editing.
    - S. Dustdar: Supervision, funding acquisition, and writing – review & editing.

4. T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polaris Scheduler: SLO- and Topology-aware Microservices Scheduling at the Edge," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022.

    - T. Pusztai: Conceptualization, investigation, methodology, software, visualization, writing – original draft, and writing – review & editing.
    - S. Nastic: Conceptualization, methodology, validation, and writing – review & editing.
    - A. Morichetta: Validation.
    - V. C. Pujol: Validation.
    - P. Raith: Validation.
    - S. Dustdar: Supervision, validation, and funding acquisition.
    - D. Vij: Conceptualization and validation.
    - Y. Xiong: Conceptualization and validation.
    - Z. Zhang: Conceptualization, methodology, and validation.

5. T. Pusztai, S. Nastic, P. Raith, S. Dustdar, D. Vij, and Y. Xiong, "Vela: A 3-Phase Distributed Scheduler for the Edge-Cloud Continuum," in *2023 IEEE International Conference on Cloud Engineering (IC2E)*, 2023.

    - T. Pusztai: Conceptualization, investigation, methodology, software, visualization, writing – original draft, and writing – review & editing.
    - S. Nastic: Conceptualization, methodology, validation, and writing – review & editing.
    - P. Raith: Conceptualization and methodology.
    - S. Dustdar: Supervision, validation, and funding acquisition.
    - D. Vij: Methodology and validation.
    - Y. Xiong: Methodology and validation.

6. T. Pusztai, C. Marcelino, and S. Nastic, "HyperDrive: Scheduling Serverless Functions in the Edge-Cloud-Space 3D Continuum," in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*, 2024.

    - T. Pusztai: Conceptualization, investigation, methodology, software, writing – original draft, and writing – review & editing.

- C. Marcelino: Conceptualization, investigation, methodology, visualization, writing – original draft, and writing – review & editing.
- S. Nastic: Supervision, conceptualization, funding acquisition, methodology, validation, and writing – review & editing.

7. T. Pusztai and S. Nastic, “ChunkFunc: Dynamic SLO-aware Configuration of Serverless Functions,” *IEEE Transactions on Parallel and Distributed Systems*, 2025.

    - T. Pusztai: Conceptualization, investigation, methodology, software, visualization, writing – original draft, and writing – review & editing.
    - S. Nastic: Supervision, conceptualization, funding acquisition, methodology, validation, and writing – review & editing.

List of Figures

List of Tables

List of Algorithms

Acronyms

AR augmented reality. 1, 95, 112

BO Bayesian Optimization. 116, 118, 119, 123–125, 128, 130, 134, 136, 137, 139, 170–173, 180, 182, 195, 197

CA Cluster Autoscaler. 164

CI Continuous Integration. 36

CLI Command-Line Interface. 26, 28–30, 32, 34, 36, 37, 42, 45, 46, 48, 50, 179

CMS Content Management System. 13

CNN Convolutional Neural Network. 95

CRD Custom Resource Definition. 28, 36, 37, 45, 78, 129

DAG directed acyclic graph. 78, 119, 126, 131, 137, 152, 157, 167, 171

DB database. 32–36, 40–42, 44–48, 50

DL Deep Learning. 172

DNN Deep Neural Networks. 172

E2E end-to-end. 108, 110, 144, 158–160, 162, 169

EI Expected Improvement. 123, 124

GA genetic algorithm. 166, 167

GCF Google Cloud Functions. 130

GCP Google Cloud Platform. 105

GP Gaussian Process. 119, 123, 124, 138, 139, 171, 182, 184

HPA Horizontal Pod Autoscaler. 14, 15, 33, 164

ILP Integer Linear Programming. 52, 56, 63, 166, 173

IoT Internet of Things. 4, 64, 93

ISL inter-satellite link. 141, 145–147, 151

KPI Key Performance Indicator. 12

LEO low earth orbit. 157, 162, 180, 183

MAPE Monitor Analyze Plan Execute. 31

MCDM multi-criteria decision making. 77, 91, 99, 144, 149, 152, 154, 165, 167, 181

ML Machine Learning. 2, 95

MRT maximum response time. 117, 122, 126, 127, 131, 132, 134, 136, 193

ms milliseconds. 49, 64–70

POI Probability of Improvement. 123, 124

Polaris Polaris SLO Cloud. 12, 15, 23, 29, 32–37, 39, 40, 42–50, 192

QoS Quality of Service. 4, 5, 7, 8, 51, 73, 74, 76, 78–81, 83, 85–87, 90, 91, 141, 152, 154, 165, 167, 180–182, 184, 192

RMSE root mean square error. 123, 124

SaaS Software-as-a-Service. 33

SDPS scheduling decisions per second. 111, 112

SLA Service Level Agreement. 2, 164, 165, 167

SLO Service Level Objective. 2–8, 11–19, 22–40, 42–51, 73–77, 84–88, 90, 91, 115–139, 141, 143, 147–152, 154, 155, 158–175, 179–185, 192, 193, 195

vCPU virtual CPU core. 76, 84, 115, 116

VM virtual machine. 48

VPA Vertical Pod Autoscaler. 164

Bibliography

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for internet of things: a primer," *Digital Communications and Networks*, 2018.

[3] Airbus, "Airbus built sentinel-2c satellite successfully launched," 2024. [Online]. Available: https://www.airbus.com/en/newsroom/press-releases/2024-09-airbus-built-sentinel-2c-satellite-successfully-launched

[4] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 472–479.

[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[6] Amazon Web Services, Inc., "5 9s (99.999%) or higher scenario with a recovery time under 1 minute," 2020. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/s-99.999-or-higher-scenario-with-a-recovery-time-under-1-minute.html

[7] ——, "Aws auto scaling features," 2020. [Online]. Available: https://aws.amazon.com/autoscaling/features/

[8] ——, "Managing throughput capacity automatically with dynamodb auto scaling," 2020. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html

[9] ——, “Aws compute optimizer,” 2023. [Online]. Available: https://aws.amazon.com/compute-optimizer/

[10] ——, “Aws lambda,” 2023. [Online]. Available: https://aws.amazon.com/lambda/

[11] ——, “Configure lambda function memory,” 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html

[12] ——, “Configure lambda function timeout,” 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-timeout.html

[13] Apache Software Foundation, “Powered by mesos: Organizations using mesos,” 2022. [Online]. Available: https://mesos.apache.org/documentation/latest/powered-by-mesos/

[14] A. Aral, I. Brandic, R. B. Uriarte, R. de Nicola, and V. Scoca, “Addressing application latency requirements through edge scheduling,” *Journal of Grid Computing*, vol. 17, no. 4, pp. 677–698, 2019.

[15] J. Arents, V. Abolins, J. Judvaitis, O. Vismanis, A. Oraby, and K. Ozols, “Human–robot collaboration trends and safety aspects: A systematic review,” *Journal of Sensor and Actuator Networks*, vol. 10, no. 3, 2021. [Online]. Available: https://www.mdpi.com/2224-2708/10/3/48

[16] M. M. Azari, S. Solanki, S. Chatzinotas, O. Kodheli, H. Sallouha, A. Colpaert, J. F. Mendoza Montoya, S. Pollin, A. Haqiqatnejad, A. Mostaani, E. Lagunas, and B. Ottersten, “Evolution of non-terrestrial networks from 5g to 6g: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2633–2672, 2022.

[17] D. Balla, C. Simon, and M. Maliosz, “Adaptive scaling of kubernetes pods,” in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–5.

[18] D. Bermbach, J. Bader, J. Hasenburg, T. Pfandzelter, and L. Thamsen, “Auctionwhisk: Using an auction-inspired approach for function placement in serverless fog platforms,” *Software: Practice and Experience*, vol. 52, no. 5, pp. 1143–1169, 2022.

[19] D. Bhattacherjee, W. Aqeel, I. N. Bozkurt, A. Aguirre, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. Maggs, and A. Singla, “Gearing up for the 21st century space race,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 113–119.

[20] D. Bhattacherjee, S. Kassing, M. Licciardello, and A. Singla, “In-orbit computing: An outlandish thought experiment?” in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 197–204.

[21] D. Bhattacherjee and A. Singla, “Network topology design at 27,000 km/hour,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 341–354.

[22] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, “Apollo: Scalable and coordinated scheduling for cloud-scale computing,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 285–300. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin

[23] Broadcom, “Using rabbitmq cluster kubernetes operator.” [Online]. Available: https://www.rabbitmq.com/kubernetes/operator/using-operator.html

[24] A. Brogi, S. Forti, C. Guerrero, and I. Lera, “How to place your apps in the fog: State of the art and open challenges,” *Software: Practice and Experience*, vol. 50, no. 5, pp. 719–740, 2020.

[25] A. Brogi, S. Forti, and A. Ibrahim, “Predictive analysis to support fog application deployment,” *Fog and edge computing: principles and paradigms*, pp. 191–222, 2019.

[26] X. Bu, J. Rao, and C.-Z. Xu, “Coordinated self-configuration of virtual machines and appliances using a model-free learning approach,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 4, pp. 681–690, 2013.

[27] V. Cardellini, F. Lo Presti, M. Nardelli, and F. Rossi, “Self-adaptive container deployment in the fog: A survey,” in *Algorithmic Aspects of Cloud Computing (ALGOCLOUD 2019)*, I. Brandic, T. A. L. Genez, I. Pietri, and R. Sakellariou, Eds. Cham: Springer International Publishing, 2020, pp. 77–102.

[28] C. Carrión, “Kubernetes scheduling: Taxonomy, ongoing issues and challenges,” *ACM Comput. Surv.*, vol. 55, no. 7, 2022.

[29] A. Casalboni, “Aws lambda power tuning,” 2023. [Online]. Available: https://github.com/alexcasalboni/aws-lambda-power-tuning

[30] P. Cassará, A. Gotta, M. Marchese, and F. Patrone, “Orbital edge offloading on mega-leo satellite constellations for equal access to computing,” *IEEE Communications Magazine*, vol. 60, no. 4, pp. 32–36, 2022.

[31] Castillo Guido A. Gavilanes, Bonetto Edoardo, Brevi Daniele, Scappatura Francesco, Sheikh Anooq, and Scopigno Riccardo, “Latency assessment of an its safety application prototype for protecting crossing pedestrians,” in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, 2020, pp. 1–5.

[32] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “The rise of serverless computing,” *Commun. ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[33] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah, “A new docker swarm scheduling strategy,” in *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, 2017, pp. 112–117.

[34] C.-Y. Chen, L.-H. Shen, K.-T. Feng, L.-L. Yang, and J.-M. Wu, “Edge selection and clustering for federated learning in optical inter-leo satellite constellation,” in *2023 IEEE 34th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2023, pp. 1–6.

[35] H. Chen, M. Xiao, and Z. Pang, “Satellite-based computing networks with federated learning,” *IEEE Wireless Communications*, vol. 29, no. 1, pp. 78–84, 2022.

[36] J. Chen, F. Xu, Y. Gu, L. Chen, F. Liu, and Z. Zhou, “Harmonybatch: Batching multi-slo dnn inference with heterogeneous serverless functions,” in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, 2024, pp. 1–10.

[37] Q. Chen, J. Qian, Y. Che, Z. Lin, J. Wang, J. Zhou, L. Song, Y. Liang, J. Wu, W. Zheng, W. Liu, L. Li, F. Liu, and K. Tan, “Yuanrong: A production general-purpose serverless system for distributed applications in the cloud,” in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 843–859.

[38] Q. Chen, G. Giambene, L. Yang, C. Fan, and X. Chen, “Analysis of inter-satellite link paths for leo mega-constellation networks,” *IEEE Transactions on Vehicular Technology*, vol. 70, no. 3, pp. 2743–2755, 2021.

[39] X. Chen, B. Hopkins, H. Wang, L. O’Neill, F. Afghah, A. Razi, P. Fulé, J. Coen, E. Rowell, and A. Watts, “Wildland fire detection and monitoring using a drone-collected rgb/ir image dataset,” *IEEE Access*, vol. 10, pp. 121 301–121 317, 2022.

[40] Y. Cheng and Z. Zhou, “Autonomous resource scheduling for real-time and stream processing,” in *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2018, pp. 1181–1184.

[41] Chris Munns, “Aws re:invent 2020: What’s new in serverless,” 2020. [Online]. Available: https://youtu.be/aW5EtKHTMuQ?t=339

[42] Chunrong Yao, Wantao Liu, Weiqing Tang, and Songlin Hu, “Eais: Energy-aware adaptive scheduling for cnn inference on high-performance gpus,” *Future Generation Computer Systems*, vol. 130, pp. 253–268, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22000127

[43] G. Copil, D. Moldovan, H. Truong, and S. Dustdar, “Sybl: An extensible language for controlling elasticity in cloud applications,” in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 112–119.

[44] R. Cordingly, S. Xu, and W. Lloyd, “Function memory optimization for heterogeneous serverless platforms with cpu time accounting,” in *2022 IEEE International Conference on Cloud Engineering (IC2E)*, 2022, pp. 104–115.

[45] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, “Elasticity in cloud computing: a survey,” *annals of telecommunications - annales des télécommunications*, vol. 70, no. 7-8, pp. 289–309, 2015.

[46] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya *et al.*, “Hydra: a federated resource manager for data-center scale analytics,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 177–192.

[47] N.-N. Dao, Q.-V. Pham, D.-T. Do, and S. Dustdar, “The sky is the edge—toward mobile coverage from the sky,” *IEEE Internet Computing*, vol. 25, no. 2, pp. 101–108, 2021.

[48] Datadog, “The state of serverless,” 2023. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[49] Dave Barth, “The bright side of sitting in traffic: Crowdsourcing road congestion data,” 2009. [Online]. Available: https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html

[50] F. Davoli, C. Kourogiorgas, M. Marchese, A. Panagopoulos, and F. Patrone, “Small satellites and cubesats: Survey of structures, architectures, and protocols,” *International Journal of Satellite Communications and Networking*, vol. 37, no. 4, pp. 343–359, 2018.

[51] C. de la Torre, B. Wagner, and M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications v5.0.* Microsoft Corporation, 2020.

[52] C. J. L. de Santana, B. de Mello Alencar, and C. V. S. Prazeres, “Reactive microservices for the internet of things,” in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, C.-C. Hung and G. A. Papadopoulos, Eds. New York, NY, USA: ACM, 2019, pp. 1243–1251.

[53] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, “Hawk: Hybrid datacenter scheduling,” in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 499–510. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/delgado

[54] C. Delimitrou, D. Sanchez, and C. Kozyrakis, “Tarcil: Reconciling scheduling speed and quality in large shared clusters,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 97–110.

[55] B. Denby and B. Lucia, “Orbital edge computing: Nanosatellite constellations as a new class of computer system,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 939–954.

[56] Docker Inc., “Scheduler design,” 2017. [Online]. Available: https://github.com/docker/swarmkit/blob/master/design/scheduler.md

[57] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, “Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 467–481.

[58] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, “Principles of elastic processes,” *Internet Computing, IEEE*, vol. 15, no. 5, pp. 66–71, 2011.

[59] S. Dustdar and I. Murturi, “Towards iot processes on the edge,” in *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future: Essays Dedicated to Michael Papazoglou on the Occasion of His 65th Birthday and His Retirement*, M. Aiello, A. Bouguettaya, D. A. Tamburri, and W.-J. van den Heuvel, Eds. Cham: Springer International Publishing, 2021, pp. 167–178.

[60] S. Dustdar, V. C. Pujol, and P. K. Donta, “On distributed computing continuum systems,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 4092–4105, 2023.

[61] R. Eidenbenz, Y.-A. Pignolet, and A. Ryser, “Latency-aware industrial fog application orchestration with kubernetes,” in *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, 2020, pp. 164–171.

[62] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, “Sizeless: Predicting the optimal size of serverless functions,” in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 248–259.

[63] S. Eismann, J. Grohmann, E. van Eyk, N. Herbst, and S. Kounev, “Predicting the costs of serverless workflows,” in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 265–276.

[64] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," 2021. [Online]. Available: https://research.spec.org/news/2020-05-29-11-38-technical-report-on-a-review-of-serverless-use-cases-and-their-characteristics-published/

[65] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 300–312.

[66] A. Elhabbash, A. Jumagaliyev, G. S. Blair, and Y. Elkhatib, "Slo-ml: A language for service level objective modelling in multi-cloud applications," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, K. Johnson, J. Spillner, D. Klusáček, and A. Anjum, Eds. New York, NY, USA: ACM, 2019, pp. 241–250.

[67] M. Elmahallawy and T. Luo, "Optimizing federated learning in leo satellite constellations via intra-plane model propagation and sink satellite scheduling," in *ICC 2023 - IEEE International Conference on Communications*, 2023, pp. 3444–3449.

[68] ETSI, "Etsi ts 101 539-3: Intelligent transport systems (its); v2x applications; part 3: Longitudinal collision risk warning (lcrw) application requirements specification," 2013-11.

[69] European Space Agency, "Sentinel-2 operations." [Online]. Available: https://www.esa.int/Enabling_Support/Operations/Sentinel-2_operations

[70] ——, "European data relay satellite system (edrs) overview," 2024. [Online]. Available: https://connectivity.esa.int/european-data-relay-satellite-system-edrs-overview

[71] ——, "European space agency-funded projects reach new performance level in groundwork for optical leo to geo data relays," 2024. [Online]. Available: https://connectivity.esa.int/news/european-space-agencyfunded-projects-reach-new-performance-level-groundwork-optical-leo-geo-data-relays

[72] ——, "Sentinel online - glossary," 2024. [Online]. Available: https://sentinels.copernicus.eu/web/sentinel/technical-guide/sentinel-2-msi/glossary

[73] Exodus Orbitals, "Satellite-as-a-service: A new approach for space industry," 2019. [Online]. Available: https://www.exodusorbitals.com/files/whitepaper.pdf

[74] A. J. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-latency-aware fog application replicas autoscaler," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[75] F. Faticanti, F. de Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, “Throughput-aware partitioning and placement of applications in fog computing,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2436–2450, 2020.

[76] Federal Communications Commission, “Kuiper systems, llc – application for authority to deploy and operate a ka-band non-geostationary satellite orbit system – order and authorization.” [Online]. Available: https://docs.fcc.gov/public/attachments/FCC-20-102A1.pdf

[77] Y. Feng, Z. Liu, Y. Zhao, T. Jin, Y. Wu, Y. Zhang, J. Cheng, C. Li, and T. Guan, “Scaling large production clusters with partitioned synchronization,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 2021, pp. 81–97. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/feng-yihui

[78] M. M. Finckenor and K. K. de Groh, “A researcher’s guide to: Space environmental effects.” [Online]. Available: https://www.nasa.gov/science-research/for-researchers/a-researchers-guide-to-space-environmental-effects/

[79] M. Fowler, *Domain-specific languages*. Upper Saddle River, NJ: Addison-Wesley, 2010.

[80] A. Fuerst and P. Sharma, “Faascache: Keeping serverless computing alive with greedy-dual caching,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 386–400.

[81] A. Furutanpey, Q. Zhang, P. Raith, T. Pfandzelter, S. Wang, and S. Dustdar, “Fool: Addressing the downlink bottleneck in satellite computing with neural feature compression,” 2024. [Online]. Available: https://arxiv.org/abs/2403.16677

[82] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing serverless to the edge with webassembly runtimes,” in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 140–149.

[83] R. Gajanin, A. Danilenka, A. Morichetta, and S. Nastic, “Towards adaptive asynchronous federated learning for human activity recognition,” in *Proceedings of the 14th International Conference on the Internet of Things (IoT 2024)*. New York, NY, USA: ACM, 2024.

[84] Z. Georgiou, M. Symeonides, D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Streamsight: A query-driven framework for streaming analytics in edge computing,” in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, 2018, pp. 143–152.

[85] M. Ghobaei-Arani and M. Ghorbian, “Scheduling mechanisms in serverless computing,” in *Serverless Computing: Principles and Paradigms*, R. Krishnamurthi, A. Kumar, S. S. Gill, and R. Buyya, Eds. Cham: Springer International Publishing, 2023, pp. 243–273.

[86] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, “Dominant resource fairness: Fair allocation of multiple resource types,” in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, 2011.

[87] GMV, “Evolution of payload management systems for communications satellites: New challenges,” 2022. [Online]. Available: https://www.gmv.com/en/media/blog/space/evolution-payload-management-systems-communications-satellites-new-challenges

[88] Google, LLC, “Autoscaling groups of instances,” 2020. [Online]. Available: https://cloud.google.com/compute/docs/autoscaler

[89] ——, “Cloud functions,” 2023. [Online]. Available: https://cloud.google.com/functions

[90] ——, “Cloud functions pricing,” 2024. [Online]. Available: https://cloud.google.com/functions/pricing

[91] V. Goronjic and S. Nastic, “Miso: A crdt-based middleware for stateful objects in the serverless edge-cloud continuum,” in *2024 IEEE International Conference on Cloud Engineering (IC2E)*, 2024, pp. 55–65.

[92] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya, “An application placement technique for concurrent iot applications in edge and fog computing environments,” *IEEE Transactions on Mobile Computing*, vol. 20, no. 4, pp. 1298–1311, 2021.

[93] J. Guan, Q. Zhang, I. Murturi, P. K. Donta, S. Dustdar, and S. Wang, “Collaborative inference in dnn-based satellite systems with dynamic task streams,” 2023. [Online]. Available: https://arxiv.org/abs/2311.06073

[94] C. Guerrero, I. Lera, and C. Juiz, “Genetic algorithm for multi-objective optimization of container allocation in cloud architecture,” *Journal of Grid Computing*, vol. 16, no. 1, pp. 113–135, 2018.

[95] Z. Han, H. Tan, X.-Y. Li, S. H.-C. Jiang, Y. Li, and F. C. M. Lau, “Ondisc: Online latency-sensitive job dispatching and scheduling in heterogeneous edge-clouds,” *IEEE/ACM Transactions on Networking*, vol. 27, no. 6, pp. 2472–2485, 2019.

[96] M. Handley, “Delay is not an option: Low latency routing in space,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 85–91.

[97] HashiCorp, “Scheduling in nomad.” [Online]. Available: https://www.nomadproject.io/docs/internals/scheduling/scheduling

[98] A. M. Hein and C. Rosete, “Space-as-a-service: A framework and taxonomy of -as-a-service concepts for space,” in *73rd International Astronautical Congress (IAC)*. International Astronautical Federation (IAF), 2022.

[99] C. Henry, “Fcc oks lower orbit for some starlink satellites,” *Space News*, 2019. [Online]. Available: https://spacenews.com/fcc-oks-lower-orbit-for-some-starlink-satellites/

[100] 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 *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, 2011.

[101] T. A. Hjeltnes and B. Hansson, “Cost effectiveness and cost efficiency in e-learning,” *QUIS-Quality, Interoperability and Standards in e-learning, Norway*, 2005.

[102] M. Höyhtyä, S. Boumard, A. Yastrebova, P. Järvensivu, M. Kiviranta, and A. Anttonen, “Sustainable satellite communications in the 6g era: A european view for multilayer systems and space safety,” *IEEE Access*, vol. 10, pp. 99 973–100 005, 2022.

[103] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, “Gaia: Geo-distributed machine learning approaching lan speeds,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 629–647. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh

[104] H. Hu, F. Liu, Q. Pei, Y. Yuan, Z. Xu, and L. Wang, “λgrapher: A resource-efficient serverless system for gnn serving through graph sharing,” in *Proceedings of the ACM Web Conference 2024*, ser. WWW ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2826–2835.

[105] P. Huang, Y. Bai, F. Li, X. Ding, Q. Chen, D. Vij, Du Peng, and Y. Xiong, “Arktos: A hyperscale cloud infrastructure for building distributed cloud,” in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022.

[106] Z. Huang, K.-J. Lin, S.-Y. Yu, and J. Yung-jen Hsu, “Co-locating services in iot systems to minimize the communication energy cost,” *Journal of Innovation in Digital Ecosystems*, vol. 1, no. 1, pp. 47–57, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352664515000061

[107] IBM Corp., “Ibm cloud functions,” 2023. [Online]. Available: https://cloud.ibm.com/functions/

[108] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, “Machine learning-based autoscaling for containerized applications,” *Neural Computing and Applications*, vol. 32, no. 13, pp. 9745–9760, 2020.

[109] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, “Predictive auto-scaling of multi-tier applications using performance varying cloud resources,” *IEEE Transactions on Cloud Computing*, vol. 10, no. 1, pp. 595–607, 2022.

[110] M. R. Jabbarpour, B. Javadi, P. Leong, R. N. Calheiros, D. Boland, and C. Butler, “Performance analysis of federated learning in orbital edge computing,” in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, ser. UCC ’23. New York, NY, USA: Association for Computing Machinery, 2024.

[111] J. Jarachanthan, L. Chen, F. Xu, and B. Li, “Astra: Autonomous serverless analytics with cost-efficiency and qos-awareness,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 756–765.

[112] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, “Container orchestration engines: A thorough functional and performance comparison,” in *2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[113] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, “The limit of horizontal scaling in public clouds,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 5, no. 1, 2020.

[114] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt, “Estimating the capacities of function-as-a-service functions,” in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC ’21. New York, NY, USA: Association for Computing Machinery, 2021.

[115] Jing Hui Alex Neo and Lee Bu Sung, “Epsilon: A microservices based distributed scheduler for kubernetes cluster,” in *2021 18th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2021, pp. 1–6.

[116] D. R. Jones, “A taxonomy of global optimization methods based on response surfaces,” *Journal of Global Optimization*, vol. 21, no. 4, pp. 345–383, 2001.

[117] D. R. Jones, M. Schonlau, and W. J. Welch, “Efficient global optimization of expensive black-box functions,” *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[118] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, “Hermod: Principled and practical scheduling for serverless functions,” in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 289–305.

[119] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 485–497. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos

[120] K. Karanasos, A. Suresh, and C. Douglas, "Advancements in yarn resource manager," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds. Cham: Springer International Publishing, 2018, pp. 1–9.

[121] T. M. Kebedew, V. N. Ha, E. Lagunas, J. Grotz, and S. Chatzinotas, "Qoe-aware cost-minimizing capacity renting for satellite-as-a-service enabled multiple-beam satcom systems," *IEEE Transactions on Communications*, vol. 72, no. 3, pp. 1773–1789, 2024.

[122] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.

[123] Kernel.org, "Energy-aware scheduling," 2024. [Online]. Available: https://docs.kernel.org/scheduler/sched-energy.html

[124] I. K. Kim, W. Wang, Y. Qi, and M. Humphrey, "Empirical evaluation of workload forecasting techniques for predictive cloud resource scaling," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016, pp. 1–10.

[125] E. Kulu, "Satellite constellations - 2021 industry survey and trends," in *35th Annual Small Satellite Conference*, 2021.

[126] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise," *Journal of Basic Engineering*, vol. 86, no. 1, pp. 97–106, 1964.

[127] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.

[128] Z. Lai, H. Li, Y. Deng, Q. Wu, J. Liu, Y. Li, J. Li, L. Liu, W. Liu, and J. Wu, "Starrynet: Empowering researchers to evaluate futuristic integrated space and terrestrial networks," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, 2023, pp. 1309–1324. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/lai-zeqi

[129] C. Li, Y. Zhang, R. Xie, X. Hao, and T. Huang, "Integrating edge computing into low earth orbit satellite networks: Architecture and prototype," *IEEE Access*, vol. 9, pp. 39 126–39 137, 2021.

[130] K. Li and S. Nastic, "Attentionfunc: Balancing faas compute across edge-cloud continuum with reinforcement learning," in *The 13th International Conference on the Internet of Things (IoT 2023)*, 2023.

[131] M. Li, J. Zhang, J. Lin, Z. Chen, and X. Zheng, "Fireface: Leveraging internal function features for configuration of functions on serverless edge platforms," *Sensors*, vol. 23, no. 18, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/18/7829

[132] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2023.

[133] Z. Li, L. O'Brien, H. Zhang, and R. Cai, "On a catalogue of metrics for evaluating commercial cloud services," in *2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE, 20.09.2012 - 23.09.2012, pp. 164–173.

[134] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, pp. 1–34, 2022.

[135] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2021.

[136] F. Liu and Y. Niu, "Demystifying the cost of serverless computing: Towards a win-win deal," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 1, pp. 59–72, 2024.

[137] J. Liu, W. Jiang, H. Han, M. He, and W. Gu, "Satellite internet of things for smart agriculture applications: A case study of computer vision," in *2023 20th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2023, pp. 66–71.

[138] X. Ma, M. Xu, Q. Li, Y. Li, A. Zhou, and S. Wang, "Visions of edge computing in 6g," in *5G Edge Computing: Technologies, Applications and Future Visions*. Singapore: Springer Nature Singapore, 2024, pp. 179–202.

[139] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, 2022, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/mahgoub

[140] F. Malandrino, C. F. Chiasserini, and G. M. Dell'Aera, "Edge-powered assisted driving for connected cars," *IEEE Transactions on Mobile Computing*, p. 1, 2021.

[141] Manner Johannes, Endreß Martin, Heckel Tobias, and Wirtz Guido, “Cold start influencing factors in function as a service,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.

[142] E. Manoel, M. J. Nielsen, A. Salahshour, S. Sampath K.V.L., and S. Sudarshanan, *Problem determination using self-managing autonomic technology*, 1st ed., ser. IBM redbooks. Austin Tex.: IBM International Technical Support Organization, 2005.

[143] C. Marcelino and S. Nastic, “Cwasi: A webassembly runtime shim for inter-function communication in the serverless edge-cloud continuum,” in *2023 IEEE/ACM Symposium on Edge Computing (SEC)*, 2023, pp. 158–170.

[144] C. Marcelino, J. Shahhoud, and S. Nastic, “Goldfish: Serverless actors with short-term memory state for the edge-cloud continuum,” in *Proceedings of the 14th International Conference on the Internet of Things*, ser. IoT ’24. New York, NY, USA: Association for Computing Machinery, 2024.

[145] Martin Fowler, “Fluentinterface,” 2005. [Online]. Available: https://martinfowler.com/bliki/FluentInterface.html

[146] G. Mateo-Garcia, J. Veitch-Michaelis, C. Purcell, N. Longepe, S. Reid, A. Anlind, F. Bruhn, J. Parr, and P. P. Mathieu, “In-orbit demonstration of a re-trainable machine learning payload for processing optical imagery,” *Scientific Reports*, vol. 13, no. 1, p. 10391, 2023.

[147] B. Matthiesen, N. Razmi, I. Leyva-Mayorga, A. Dekorsy, and P. Popovski, “Federated learning in satellite constellations,” *IEEE Network*, vol. 38, no. 2, pp. 232–239, 2024.

[148] T. Menouer, C. Cérin, and É. Leclercq, “New multi-objectives scheduling strategies in docker swarmkit,” in *Algorithms and Architectures for Parallel Processing*, J. Vaidya and J. Li, Eds. Cham: Springer International Publishing, 2018, pp. 103–117.

[149] Microsoft, “Linux virtual machines pricing | microsoft azure.” [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

[150] ——, “Autoscaling,” 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling

[151] ——, “Azure functions,” 2023. [Online]. Available: https://azure.microsoft.com/en-us/products/functions

[152] V. Millnert and J. Eker, “Holoscale: horizontal and vertical scaling of cloud resources,” in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 196–205.

[153] D. Milojicic, "The edge-to-cloud continuum," *Computer*, vol. 53, no. 11, pp. 16–25, 2020.

[154] J. Mockus, V. Tiesis, and A. Zilinskas, "The application of bayesian methods for seeking the extremum, vol. 2," *L Dixon and G Szego. Toward Global Optimization*, vol. 2, 1978.

[155] P. Mohagheghi and Ø. Haugen, "Evaluating domain-specific modelling solutions," in *Advances in Conceptual Modeling – Applications and Challenges*, ser. Lecture Notes in Computer Science, J. Trujillo, G. Dobbie, H. Kangassalo, S. Hartmann, M. Kirchberg, M. Rossi, I. Reinhartz-Berger, E. Zimányi, and F. Frasincar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6413, pp. 212–221.

[156] N. Mohan, A. E. Ferguson, H. Cech, R. Bose, P. R. Renatin, M. K. Marina, and J. Ott, "A multifaceted look at starlink performance," in *Proceedings of the ACM on Web Conference 2024*, ser. WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2723–2734.

[157] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Mela: elasticity analytics for cloud services," *International Journal of Big Data Intelligence*, vol. 2, no. 1, pp. 45–62, 2015.

[158] A. Morichetta, V. C. Pujol, S. Nastic, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, "Polarisprofiler: A novel metadata-based profiling approach for optimizing resource management in the edge-cloud continnum," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2023, pp. 27–36.

[159] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, "ifogstor: An iot data placement strategy for fog infrastructure," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, 2017, pp. 97–104.

[160] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly, 2016.

[161] NanoAvionics, "Startical to test its technology on a nanoavionics-built satellite, paving the way for the first space-based air traffic service constellation," 2024. [Online]. Available: https://nanoavionics.com/news/startical-to-test-its-technology-on-a-nanoavionics-built-satellite-paving-the-way-for-the-first-space-based-air-traffic-service-constellation/

[162] S. Nastic, "Self-provisioning infrastructures for the next generation serverless computing," *SN Computer Science*, vol. 5, no. 6, pp. 678–693, 2024.

[163] S. Nastic, G. Copil, H.-L. Truong, and S. Dustdar, "Governing elastic iot cloud systems under uncertainty," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015, pp. 131–138.

[164] S. Nastic, A. Morichetta, T. Pusztai, S. Dustdar, X. Ding, D. Vij, and Y. Xiong, "Sloc: Service level objectives for next generation cloud computing," *IEEE Internet Computing*, vol. 24, no. 3, pp. 39–50, 2020.

[165] S. Nastic, T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, D. Vij, and Y. Xiong, "Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 206–216.

[166] S. Nastic, P. Raith, A. Furutanpey, T. Pusztai, and S. Dustdar, "A serverless computing fabric for edge & cloud," in *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, 2022, pp. 1–12.

[167] National Aeronautics and Space Administration, "State-of-the-art small spacecraft technology." [Online]. Available: https://www.nasa.gov/wp-content/uploads/2024/03/soa-2023.pdf

[168] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors (Basel, Switzerland)*, vol. 20, no. 16, 2020.

[169] H. Ning, H. Wang, Y. Lin, W. Wang, S. Dhelim, F. Farha, J. Ding, and M. Daneshmand, "A survey on the metaverse: The state-of-the-art, technologies, applications, and challenges," *IEEE Internet of Things Journal*, 2023.

[170] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 198–206.

[171] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Assessing architecture conformance to coupling-related patterns and practices in microservices," in *Software Architecture*, ser. Lecture Notes in Computer Science, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, Eds. Cham: Springer International Publishing, 2020, vol. 12292, pp. 3–20.

[172] Orbit.ing-now.com, "Low earth orbit," 2024. [Online]. Available: https://orbit.ing-now.com/low-earth-orbit/

[173] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 69–84.

[174] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 13–26.

[175] S. Pallewatta, V. Kostakos, and R. Buyya, “Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments,” in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 71–81.

[176] L. Pan, L. Wang, S. Chen, and F. Liu, “Retention-aware container caching for serverless edge computing,” in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1069–1078.

[177] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, “Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions,” in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 324–340.

[178] T. Pfandzelter, “Serverless abstractions for edge computing in large low-earth orbit satellite networks,” in *Proceedings of the 24th International Middleware Conference: Demos, Posters and Doctoral Symposium*, ser. Middleware ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 3–6.

[179] T. Pfandzelter and D. Bermbach, “Qos-aware resource placement for leo satellite edge computing,” in *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, 2022, pp. 66–72.

[180] T. Pfandzelter, J. Hasenburg, and D. Bermbach, “Towards a computing platform for the leo edge,” in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 43–48.

[181] N. Potu, C. Jatoth, and P. Parvataneni, “Optimizing resource scheduling based on extended particle swarm optimization in fog computing environments,” *Concurrency and Computation: Practice and Experience*, vol. 33, no. 23, 2021.

[182] T. Pusztai, C. Marcelino, and S. Nastic, “Hyperdrive: Scheduling serverless functions in the edge-cloud-space 3d continuum,” in *2024 IEEE/ACM Symposium on Edge Computing (SEC)*, 2024.

[183] T. Pusztai, A. Morichetta, V. C. Pujol, S. Dustdar, S. Nastic, X. Ding, D. Vij, and Y. Xiong, “A novel middleware for efficiently implementing complex cloud-native slos,” in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 410–420.

[184] ——, “Slo script: A novel language for implementing complex cloud-native elasticity-driven slos,” in *2021 IEEE International Conference on Web Services (ICWS)*, 2021, pp. 21–31.

[185] T. Pusztai and S. Nastic, “Chunkfunc: Dynamic slo-aware configuration of serverless functions,” *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[186] T. Pusztai, S. Nastic, A. Morichetta, V. Casamayor Pujol, P. Raith, S. Dustdar, D. Vij, Y. Xiong, and Z. Zhang, “Polaris scheduler: Slo- and topology-aware microservices scheduling at the edge,” in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, 2022.

[187] T. Pusztai, S. Nastic, P. Raith, S. Dustdar, D. Vij, and Y. Xiong, “Vela: A 3-phase distributed scheduler for the edge-cloud continuum,” in *2023 IEEE International Conference on Cloud Engineering (IC2E)*, 2023.

[188] T. Pusztai, F. Rossi, and S. Dustdar, “Pogonip: Scheduling asynchronous applications on the edge,” in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 660–670.

[189] X. Qiao, P. Ren, S. Dustdar, L. Liu, H. Ma, and J. Chen, “Web ar: A promising future for mobile augmented reality—state of the art, challenges, and insights,” *Proceedings of the IEEE*, vol. 107, no. 4, pp. 651–666, 2019.

[190] X. Qiu, W. Zhang, W. Chen, and Z. Zheng, “Distributed and collective deep reinforcement learning for computation offloading: A practical perspective,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1085–1101, 2021.

[191] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-scaling web applications in clouds,” *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–33, 2018.

[192] RAINBOW Project, “D1.1 – rainbow stakeholders requirements analysis,” 2020. [Online]. Available: https://rainbow-h2020.eu/deliverables/

[193] ——, “D1.3 – rainbow use-cases descriptions,” 2021. [Online]. Available: https://rainbow-h2020.eu/deliverables/

[194] P. Raith, S. Nastic, and S. Dustdar, “Serverless edge computing—where we are and what lies ahead,” *IEEE Internet Computing*, vol. 27, no. 3, pp. 50–64, 2023.

[195] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, “Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 33–40.

[196] T. Rausch, C. Lachner, P. A. Frangoudis, P. Raith, and S. Dustdar, “Synthesizing plausible infrastructure configurations for evaluating edge computing systems,” in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/hotedge20/presentation/rausch

[197] T. Rausch, A. Rashed, and S. Dustdar, “Optimized container scheduling for data-intensive serverless edge computing,” *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X2030399X

[198] A. Raza, N. Akhtar, V. Isahagian, I. Matta, and L. Huang, “Configuration and placement of serverless applications using statistical learning,” *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1065–1077, 2023.

[199] M. A. Rodriguez and R. Buyya, “Container–based cluster orchestration systems: A taxonomy and future directions,” *Software: Practice and Experience*, vol. 49, no. 5, pp. 698–719, 2019.

[200] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, “Geo-distributed efficient deployment of containers with kubernetes,” *Computer Communications*, vol. 159, pp. 161–174, 2020.

[201] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 329–338.

[202] R. B. Roy, T. Patel, and D. Tiwari, “Icebreaker: Warming serverless functions better with heterogeneity,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 753–767.

[203] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, “Autopilot: workload autoscaling at google,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, A. Bilas, K. Magoutis, E. Markatos, D. Kostic, and M. Seltzer, Eds. New York, NY, USA: ACM, 2020, pp. 1–16.

[204] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, “Slam: Slo-aware memory optimization for serverless applications,” in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 30–39.

[205] Santos José, Wauters Tim, Volckaert Bruno, and De Turck Filip, “Towards network-aware resource provisioning in kubernetes for fog computing applications,” in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 351–359.

[206] Satellogic, “Constellation-as-a-service,” 2025. [Online]. Available: https://satellogic.com/products/constellation-as-a-service/

[207] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[208] E. Saurez, H. Gupta, A. Daglis, and U. Ramachandran, “Oneedge: An efficient control plane for geo-distributed infrastructures,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 182–196.

[209] M. Schwarzkopf, “The evolution of cluster scheduler architectures,” 2016. [Online]. Available: https://www.cl.cam.ac.uk/research/srg/netos/camsas/blog/2016-03-09-scheduler-architectures.html

[210] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: Flexible, scalable schedulers for large compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 351–364.

[211] B. Sedlak, I. Murturi, and S. Dustdar, “Specification and operation of privacy models for data streams on the edge,” in *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, 2022, pp. 78–82.

[212] S. Sehic, F. Li, S. Nastic, and S. Dustdar, “A programming model for context-aware applications in large-scale pervasive systems,” in *Proceedings of the IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2012).* IEEE Computer Society, 2012, pp. 142–149.

[213] P. Senior, S. Eckersley, V. Irwin, B. Stern, A. Haslehurst, A. Cawthorne, A. Da Silva Curiel, and M. Sweeting, “Can we use low cost small satellites to observe space debris missed by ground systems,” in *Proceedings of the 8th European Conference on Space Debris, Darmstadt, Germany*, 2021, pp. 20–23.

[214] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural implications of function-as-a-service computing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1063–1075.

[215] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[216] D. Shepardson, “Fcc chair wants more competition to spacex’s starlink unit,” 2024. [Online]. Available: https://www.reuters.com/technology/space/fcc-chair-wants-more-competition-spacexs-starlink-unit-2024-09-11/

[217] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *2020 USENIX Annual Technical Conference*

(USENIX ATC 20). USENIX Association, 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[218] Y. Siriwardhana, P. Porambage, M. Liyanage, and M. Ylianttila, "A survey on mobile augmented reality with 5g mobile edge computing: Architectures, applications, and technical aspects," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1160–1192, 2021.

[219] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized iot service placement in the fog," *Service Oriented Computing and Applications*, vol. 11, pp. 427–443, 2017.

[220] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards qos-aware fog service placement," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 14.05.2017 - 15.05.2017, pp. 89–96.

[221] H. Song, F. Chauvel, and P. H. Nguyen, "Using microservices to customize multi-tenant software-as-a-service," in *Microservices: Science and Engineering*. Springer, 2020, pp. 299–331.

[222] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith, "Runtime vertical scaling of virtualized applications via online model estimation," in *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems*, 2014, pp. 157–166.

[223] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 19–24.

[224] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Demo: Emulating geo-distributed fog services," in *Proceedings of the 5th ACM/IEEE Symposium on Edge Computing*, ser. SEC '20. New York, NY, USA: Association for Computing Machinery, 2020.

[225] ——, "Fogify: A fog computing emulation framework," in *Proceedings of the 5th ACM/IEEE Symposium on Edge Computing*, ser. SEC '20. New York, NY, USA: Association for Computing Machinery, 2020.

[226] Q. Tang, R. Xie, Z. Fang, T. Huang, T. Chen, R. Zhang, and F. R. Yu, "Joint service deployment and task scheduling for satellite edge computing: A two-timescale hierarchical approach," *IEEE Journal on Selected Areas in Communications*, vol. 42, no. 5, pp. 1063–1079, 2024.

[227] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig, "rsla: A service level agreement language for cloud services," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 27.06.2016 - 02.07.2016, pp. 415–422.

[228] The Apache Software Foundation, "Apache hadoop 3.3.3: Capacity scheduler." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[229] ——, "Apache hadoop 3.3.3: Fair scheduler." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[230] The Kubernetes Authors, "Custom metrics api - design proposal," 2018-01-22. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/custom-metrics-api.md

[231] ——, "Hpa v2 api extension proposal," 2018-02-14. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/hpa-external-metrics.md

[232] ——, "Horizontal pod autoscaler with arbitrary metrics - design proposal," 2018-11-19. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/hpa-v2.md

[233] ——, "External metrics api - design proposal," 2018-12-14. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/external-metrics-api.md

[234] ——, "Autoscaling components for kubernetes," 2020. [Online]. Available: https://github.com/kubernetes/autoscaler

[235] ——, "Scheduler configuration | kubernetes," 2022. [Online]. Available: https://kubernetes.io/docs/reference/scheduling/config/

[236] ——, "Considerations for large clusters," 2023-01-12. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/

[237] ——, "Scheduling framework | kubernetes," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[238] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and H. Yang, "Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 78–93.

[239] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, vol. 21, no. 3, pp. 940–954, 2022.

[240] A. Ullah, J. Li, Y. Shen, and A. Hussain, "A control theoretical view of cloud elasticity: taxonomy, survey and challenges," *Cluster Computing*, vol. 21, no. 4, pp. 1735–1764, 2018.

[241] V. Pujol, P. Donta, A. Morichetta, I. Murturi, and S. Dustdar, “Edge intelligence—research opportunities for distributed computing continuum systems,” *IEEE Internet Computing*, vol. 27, no. 04, pp. 53–74, 2023.

[242] C. van Arsdale, “A breakthrough in wildfire detection: How a new constellation of satellites can detect smaller wildfires earlier,” 2024. [Online]. Available: https://blog.google/outreach-initiatives/sustainability/google-ai-wildfire-detection/

[243] D. Vasisht, J. Shenoy, and R. Chandra, “L2d2: low latency distributed downlink for leo satellites,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM ’21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 151–164.

[244] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC ’13. New York, NY, USA: Association for Computing Machinery, 2013.

[245] Verizon, “Ip latency statistics,” 2023. [Online]. Available: https://www.verizon.com/business/terms/latency/

[246] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the Tenth European Conference on Computer Systems - EuroSys ’15*, L. Réveillère, T. Harris, and M. Herlihy, Eds. New York, New York, USA: ACM Press, 2015, pp. 1–17.

[247] V. Villani, F. Pini, F. Leali, and C. Secchi, “Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications,” *Mechatronics*, vol. 55, pp. 248–266, 2018.

[248] C. Wang, Y. Zhang, Q. Li, A. Zhou, and S. Wang, “Satellite computing: A case study of cloud-native satellites,” in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023, pp. 262–270.

[249] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[250] M. Wang, D. Zhang, and B. Wu, “A cluster autoscaler based on multiple node types in kubernetes,” in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 12.06.2020 - 14.06.2020, pp. 575–579.

[251] S. Wang and Q. Li, “Satellite computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 10, no. 24, pp. 22 514–22 529, 2023.

[252] W. Wang, H. Chen, and X. Chen, “An availability-aware virtual machine placement approach for dynamic scaling of cloud applications,” in *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*, 2012, pp. 509–516.

[253] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, “Fog orchestration for internet of things services,” *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, 2017.

[254] Z. Wen, Q. Chen, Y. Niu, Z. Song, Q. Deng, and F. Liu, “Joint optimization of parallelism and resource configuration for serverless function steps,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, pp. 560–576, 2024.

[255] R. Xing, X. Ma, A. Zhou, S. Dustdar, and S. Wang, “From earth to space: A first deployment of 5g core network on satellite,” *China Communications*, vol. 20, no. 4, pp. 315–325, 2023.

[256] F. Xu, Y. Qin, L. Chen, Z. Zhou, and F. Liu, “λdnn: Achieving predictable distributed dnn training with serverless architectures,” *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 450–463, 2022.

[257] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, “On the use of fuzzy modeling in virtualized data center management,” in *Fourth International Conference on Autonomic Computing (ICAC’07)*, 2007, p. 25.

[258] L. Yazdanov and C. Fetzer, “Vscaler: Autonomic virtual machine scaling,” in *2013 IEEE Sixth International Conference on Cloud Computing*, 2013, pp. 212–219.

[259] G. Yu, P. Chen, Z. Zheng, J. Zhang, X. Li, and Z. He, “Faasdeliver: Cost-efficient and qos-aware function delivery in computing continuum,” *IEEE Transactions on Services Computing*, pp. 1–16, 2023.

[260] H. Yu, C. Fontenot, H. Wang, J. Li, X. Yuan, and S.-J. Park, “Libra: Harvesting idle resources safely and timely in serverless clusters,” in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 181–194.

[261] H. Yu, A. A. Irissappane, H. Wang, and W. J. Lloyd, “Faasrank: Learning to schedule functions in serverless platforms,” in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2021, pp. 31–40.

[262] H. W. Zaglauer, “Intelligent satellite payloads as enablers for 6g,” ETSI Conference on Non-Terrestrial Networks, A Native Component of 6G, 2024. [Online]. Available: https://docbox.etsi.org/Workshop/2024/04_ETSI_6G_NTN/SESSION%2006/S6_03_Zaglauer.pdf

[263] Z. Zhai, L. Zeng, T. Ouyang, S. Yu, Q. Huang, and X. Chen, “Seco: Multi-satellite edge computing enabled wide-area and real-time earth observation missions,” in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, 2024, pp. 2548–2557.

[264] W. Zhang, Y. Xue, J. Wu, and X. Xu, “Satellite as a service: a hybrid resource management framework for space-terrestrial integrated networks,” in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 2020, pp. 171–174.

[265] X. Zhang, J. Liu, R. Zhang, Y. Huang, J. Tong, N. Xin, L. Liu, and Z. Xiong, “Energy-efficient computation peer offloading in satellite edge computing networks,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 4, pp. 3077–3091, 2024.

[266] Z. Zhang, C. Jin, and X. Jin, “Jolteon: Unleashing the promise of serverless for serverless workflows,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, 2024, pp. 167–183. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-jolteon

[267] Zhao Hailiang, Deng Shuiguang, Liu Zijie, Yin Jianwei, and Dustdar Schahram, “Distributed redundant placement for microservice-based applications at the edge,” *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1732–1745, 2022.

[268] D. Zhou, M. Sheng, J. Li, and Z. Han, “Aerospace integrated networks innovation for empowering 6g: A survey and future challenges,” *IEEE Communications Surveys & Tutorials*, vol. 25, no. 2, pp. 975–1019, 2023.

[269] Z. Zhou, Y. Zhang, and C. Delimitrou, “Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1–14.

[270] T. Zubko, A. Jindal, M. Chadha, and M. Gerndt, “Maff: Self-adaptive memory optimization for serverless functions,” in *Service-Oriented and Cloud Computing*, F. Montesi, G. A. Papadopoulos, and W. Zimmermann, Eds. Cham: Springer International Publishing, 2022, pp. 137–154.