Informatics

Recommendation for Orchestration Architectures in Serverless Edge Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Kian Pouresmaeil
Matrikelnummer 01529142

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Univ.Ass. Dipl.-Ing. Philipp Alexander Raith, BSc

Wien, 24. Mai 2021

Kian Pouresmaeil   Schahram Dustdar

TU WIEN Informatics

Recommendation for Orchestration Architectures in Serverless Edge Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software and Information Engineering

by

Kian Pouresmaeil
Registration Number 01529142

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Univ.Ass. Dipl.-Ing. Philipp Alexander Raith, BSc

Vienna, 24th May, 2021

Kian Pouresmaeil	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Kian Pouresmaeil

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.

Wien, 24. Mai 2021

Kian Pouresmaeil

Acknowledgements

First and foremost, I want to thank my advisors, Schahram Dustdar and my co-supervisor, Philipp Raith. I especially want to express my gratitude to Phillip for his commitment, our productive discussions, and the invaluable feedback I received.

Furthermore, I want to thank my friends and family for their unwavering support during my studies, always believing in me, cheering me up when I was feeling down, and distracting me when I needed it.

Finally, I want to thank Jasmin for always looking after me, for her immense patience, and for her endless support.

Kurzfassung

Cloud Computing kann die Arbeitslast neuer Anwendungen, wie autonomer Fahrzeuge oder Augmented Reality, nicht effizient bewältigen. Um dieses Problem zu lösen, entstand ein neues Paradigma namens Edge Computing. Dies ermöglicht die Verarbeitung in unmittelbarer Nähe der Nutzer, welche unter anderem eine geringere Latenzzeit ermöglicht, die für diese neuen Anwendungen unerlässlich ist. Trotz allem bringt Edge Computing auch neue Herausforderungen mit sich. Bei dem neuen Paradigma müssen die Entwickler mit heterogenen Hardware- und Netzwerkinfrastrukturen umgehen, welche die Bereitstellung der Anwendung und das Ressourcenmanagement erschwert. Als Lösung für diese Herausforderungen entstand ein neues Paradigma namens Serverless Edge Computing. Dabei wird von den Entwicklern eine Applikation in Form von stateless Funktionen bereitgestellt, welche dann von einem Serviceprovider gehostet werden. Dadurch übernehmen die Serviceprovider die Verantwortung für die Bereitstellung von Anwendungen und das Ressourcenmanagement. Dies bedeutet, dass Entwickler sich nicht mehr um die Heterogenität des Edge Computings kümmern müssen. Ein wichtiger Teil des idealen Ressourcenmanagements ist die Ressourcenorchestrierung, die für die effiziente Handhabung von Systemressourcen und das Erreichen von Service-Level-Zielen, wie z. B. Latenz, unerlässlich ist. In der aktuellen Literatur findet man viele verschiedene Orchestrierungsarchitekturen, was darauf hinweist, dass die Suche nach einer optimalen Strategie ein komplexes Problem darstellt.

In dieser Arbeit stellen wir ein Framework vor, welches in der Lage ist Orchestrierungsarchitekturen auf der Basis aussagekräftiger Leistungsmetriken zu empfehlen. Zu diesem Zweck erweitern wir ein serverless Experimentierframework, indem wir neue Funktionalität implementieren. Zu den neuen Funktionen gehören, unter anderem, die dynamische Erstellung heterogener Infrastrukturen mit Kubernetes-Clustern und die Unterstützung mehrerer Orchestrierungsarchitekturen. Unser Framework basiert auf einem Systemmodell, das in der Lage ist die Funktionsweise einer Orchestrierungsarchitektur auf einer heterogenen Infrastruktur darzustellen. Um die Fähigkeiten unseres Frameworks zu demonstrieren, untersuchen wir zwei verschiedene Szenarien. Dabei betrachten wir jeweils eine zentrale, eine dezentrale und eine verteilte Orchestrierungsarchitektur. Die Szenarien basieren auf bekannten Anwendungsfällen für Edge-Cloud-Systeme namens Smart City und Industrial IoT. Damit wir die verschiedenen Orchestrierungsstrategien aussagekräftig bewerten können, basieren wir unsere Empfehlungen auf aussagekräftige

Leistungsmetriken aus der Sicht von Benutzern und Plattformanbietern, sowie auf die Auswirkungen der Infrastruktur auf die Orchestrierungsarchitektur.

Unsere Ergebnisse zeigen, dass aus Sicht der Nutzer eine zentralisierte Architektur die beste Wahl für ein Industrial IoT-Szenario ist. Im Gegensatz dazu ist eine dezentralisierte Architektur die beste Wahl für ein Smart City-Szenario. Aus der Sicht von den beiden anderen Perspektiven, erwies sich die dezentrale Architektur in beiden Szenarien als effektiver, als die beiden anderen Architekturen.

Abstract

Cloud computing cannot efficiently handle the workload of new applications such as autonomous vehicles or augmented reality. To overcome this problem, a new paradigm named edge computing rose. It allows processing closer to users, enabling, among other things, lower latency, which is vital for these new applications. Nevertheless, edge computing introduces new challenges. In the edge computing paradigm, developers have to handle heterogeneous hardware and network infrastructures, making application deployment and resource management challenging. To overcome the deployment difficulties, another new paradigm named serverless edge computing came to life. Developers deploy their applications as stateless functions, which are hosted by a service provider. Thus, developers do not have to worry about the heterogeneous nature of edge computing, and the service provider takes over the deployment and resource management responsibility. One major part of ideal resource management is resource orchestration, which is vital for efficiently handling system resources and reaching Service Level Objectives such as latency. Current literature supports many different orchestration architectures, suggesting that finding an optimal strategy is a complex problem.

This thesis introduces a framework capable of recommending orchestration architectures based on expressive performance metrics. For this purpose, we extend a serverless experimentation framework by implementing new functionalities, such as dynamically creating heterogeneous infrastructures with Kubernetes clusters and supporting multiple orchestration architectures. Our framework is based on a system model capable of describing the operation of an orchestration architecture on a heterogeneous infrastructure. To showcase our framework's capabilities, we investigate two different scenarios using a centralized, decentralized, and distributed orchestration architecture. The scenarios are based on known edge-cloud system use cases: Smart City and Industrial IoT. To properly evaluate the different orchestration strategies, we base our recommendation on meaningful performance metrics from a user's and platform provider's perspective, as well as the impact of the infrastructure on the orchestration architecture.

Our results show that from a user's perspective, a centralized architecture is the best choice for an IIoT scenario. In contrast, a decentralized architecture is the best choice for a Smart City scenario. Considering the other two perspectives, the decentralized architecture proved more effective than the other two architectures in both scenarios.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

Due to the emergence of edge intelligence applications, new requirements for latency, geo-distribution, mobility support, and more have been introduced [74]. For example, applications such as autonomous vehicles or augmented reality require low latency. The new requirements are essential for these applications to operate smoothly. Otherwise, they would not manage to respond in real-time to changes in their environment and, therefore, make the application futile. [65, 57].

Traditional cloud data centers are geographically centralized and far from end devices/users. Therefore, using only cloud centers is not feasible for handling these challenges. To combat these weaknesses of cloud data centers, edge computing has been proposed [43].

Edge computing allows computation closer to users, enabling better Quality-of-Experience (QoE) and Quality-of-Service (QoS). However, the edge computing paradigm itself can only partially replace cloud computing. The edge cannot match cloud data centers' sheer computational power and resource capacity. Nevertheless, these technologies complement each other, achieving better QoE for users [41]. Furthermore, edge intelligence applications operate on multiple layers between edge and cloud [74]. Therefore, observing both paradigms as the edge-cloud continuum is crucial to gaining a comprehensive understanding.

Edge computing also comes with its challenges. In particular, resource management is a significant concern due to heterogeneity being a recognizable aspect in this domain. Edge computing elements use various platforms, hardware, and network technologies [27]. Moreover, among other challenges, heterogeneity plays a vital role in why deploying applications at the edge is challenging [24].

One way to help the developer in this situation is the new paradigm called serverless computing [5]. Here, the responsibility of resource management shifts away from the user and to the service provider. In serverless computing, also known as Function-as-a-Service (FaaS), developers can deploy an application as a stateless function. Initially, serverless computing was created for the cloud, but it also found its way to the edge over time. Placing serverless functions on the edge has the advantage of low latency compared to the cloud layer. Therefore, moving them to the edge is particularly suitable for latency-critical functions. However, moving to edge also means that serverless systems must work with resource-constrained hardware or, in general terms, restricted infrastructures. Therefore, serverless platforms should only add minimal overhead to avoid performance loss [9]. Moreover, function placement is NP-hard, making it a complex problem to solve [55].

As mentioned above, in serverless computing, the platform provider is responsible for proper resource management. A significant aspect of this management is resource orchestration, which refers to automated consideration of the resource requirements of workloads and efficient handling of available system resources [44]. It is paramount to address appropriate scheduling and scaling strategies to establish appropriate orchestration [44, 64]. In this context, scaling refers to allocating resources in real-time to maintain application performance requirements and QoS. For example, resources should scale up when the demand rises and scale down when the demand decreases. Scheduling aims to find suitable host nodes for workloads based on their resource requirements while maximizing the utilization of available resources [44]. The most mentioned explicit goals in current literature are improved and effective resource management or the guarantee of Service Level Objectives (SLOs) in latency reduction, application performance, or application response time [13].

Based on current serverless frameworks, it is not possible to clearly indicate, which resource orchestration approach performs best. The current literature supports different serverless frameworks for the edge-cloud continuum, following different approaches and architectures. For example, the work of Baresi et al. [6] follows an optimization driven approach, while Han et al. [21] follow an AI driven approach. Additionally, both frameworks use different architectures despite having the same goal. Further frameworks can be found in Chapter 3. As there are different approaches to solving the same problem, it implies that the most effective approach has not yet been determined, as well as it is a complex problem to solve. Furthermore, a preliminary literature research found a lack of frameworks for orchestration architecture strategy testing,

Different architectures need to be explored in greater depth. We explore a range of architectures, including centralized, decentralized, and distributed architectures. While in a centralized approach, an all-known master element is responsible for all orchestration decisions, in decentralized architectures, the orchestration decisions are divided into intra-cluster and inter-cluster decisions. Unlike centralized and decentralized architectures, distributed architectures do not have a master element, so every cluster or node makes its own decisions [30].

To this end, we create a framework for orchestration architecture evaluation based on the benchmarking tool Galileo [59]. We can use most of the components in Galileo out of the box, while the load-balancer and autoscaler need to be fine-tuned for each orchestration architecture. Furthermore, Galileo does not provide any custom scheduler, so it relies on the default Kubernetes scheduler instead. Given that various orchestration architectures have distinct scheduling approaches, we designed custom schedulers, which are also compatible with the Galileo framework. Moreover, we develop a framework that facilitates the whole process, from creating virtual machines to gathering metrics from experiments. This allows us and future users to conduct multiple different experiments with less overhead.

This research project aims to reduce the complexity required to decide when and what orchestration architecture should be used by investigating the performance of orchestration architectures on different infrastructures from different perspectives. In order to accomplish this, we implement various orchestration architectures in Kubernetes [34] and benchmark their performance based on key performance indicators (KPIs). Furthermore, we work on a level where our approach and reasoning is not constrained by the tools and technologies we use, and therefore generally applicable. For example, the framework can be extended to other orchestration platforms. Finally, we form a recommendation for orchestration architectures derived from the evaluation of our collected data.

1.2 Aim of the Thesis and Expected Results

Motivated by the problems described in Section 1.1, this thesis investigates the performance of orchestration architectures in serverless computing. Therefore, based on KPIs, we evaluate the main orchestration architecture strategies, namely centralized, distributed, and decentralized [30]. For that reason, custom orchestration components, such as schedulers, adjusted to an orchestration architecture need to be implemented.

In order to gather the necessary data, we implement the mentioned orchestration architectures in Kubernetes, and use a predefined stack of data. Benchmarking is conducted using an extension of the open-source tool Galileo, a pre-written autoscaler and load-balancer [59, 58], and scheduler. This thesis answers the following research questions:

RQ1 How can we describe and categorize heterogeneous edge-cloud systems and orchestration architectures? Nowadays, there exists numerous amount of edge-cloud systems [60]. These systems are usually heterogeneous regarding hardware, network, and more. Furthermore, there exist multiple different orchestration architectures. Considering the various scenarios, a categorization would undoubtedly be helpful to create a specific structure. A systematic description capable of modeling the heterogeneous infrastructures and orchestration architectures builds the foundation for a unified approach to evaluating such systems. This description also forms the basis for our framework. It is expected to define a categorization of edge-cloud systems, which can be used for RQ3, and a system model capable of describing heterogeneous edge-cloud systems and orchestration architectures.

RQ2 What KPIs are appropriate to evaluate an orchestration architecture? There exists a wide variety of performance metrics for the edge-cloud continuum [3]. Potential KPIs are CPU/RAM resource usage, network throughput, application response time, and scheduling duration [3, 8]. KPIs are vital to evaluate orchestration architectures objectively. It is expected to find several metrics representing the performance of an orchestration architecture.

RQ3 How can we create a system that allows and illustrates the differences between orchestration architectures and heterogeneous infrastructures and can make recommendations for those? This work aims to make recommendations considering orchestration architecture and infrastructure from the perspective of a user, a platform provider, and the impact of different infrastructures on the architectures. From the user's point of view, we recommend an architecture that can process requests as quickly as possible. From the platform provider's perspective, we concern us with the operating costs. Regarding the infrastructure impact, we want to provide a recommendation indicating which architecture is least affected by its infrastructure. The recommendation is based on the categorization from RQ1 and the obtained KPIs from RQ2. The different architectures are expected to have different impacts and perform differently in the same circumstances. We do not expect to find the most optimal solution for each architecture. Instead, we create a framework based on the system model from RQ1, which allows users to find the best solution. We are evaluating this solution based on a detailed use case study.

By answering these questions, this thesis contributes significantly to expanding knowledge about the impact of different orchestration architectures. Furthermore, it supports engineers during the design phase by simplifying the complexity required to decide the more suitable architecture in specific scenarios.

1.3 Approach

This thesis aims to create a recommendation for orchestration architectures in serverless edge computing. In order to reach this goal, we decided to use empirical experiments to gather data and use that data later to create a recommendation. The overall approach is described as follows.

Before starting the experiments, it is crucial to research existing orchestration architectures for serverless edge computing. Since our recommendation should correlate with the real world, we create infrastructures that mimic real-world edge scenarios and their constraints as closely as possible. Furthermore, we combine the different infrastructures with centralized, decentralized, and distributed orchestration architectures to investigate the performance of each orchestration architecture in different scenarios.

We must base our evaluation on impactful metrics to assess different approaches to orchestration architectures on different infrastructures. Therefore, we assess the outcome of our experiments based on expressive performance metrics relevant to edge-cloud systems found in current literature [3, 8]. The key performance indicators are categorized into four different measurement levels: system, network, application, and orchestration. This categorization should allow us to approach the evaluation in a more structured way. Moreover, it helps readers quickly identify the strengths and weaknesses of orchestration architectures.

We use our extension of the open-source tool Galileo [59] as the benchmarking tool for our experiments. Among other things, custom schedulers, tailored to an orchestration architecture, are implemented to assign the collected KPIs to an architecture more effectively. Experiments are categorized into scenarios, where each scenario consists of an orchestration architecture, a heterogeneous infrastructure, a specific user request pattern, and network settings.

We evaluate the performance of the different orchestration architectures and compare them to each other. Based on the findings of the experiments, the strengths and weaknesses of the different strategies are discovered, and a meaningful evaluation can be given. Finally, a recommendation can be formed based on the evaluation and findings during the literature research regarding edge-cloud scenarios.

1.4 Structure

The remainder of this thesis is structured as follows. Chapter 2 outlines the relevant background information of this work. In particular, we give an overview of edge intelligence in the edge-cloud continuum, serverless edge computing, and orchestration architectures and describe the tools and technologies used for our framework. Chapter 3 represents the related work, highlighting work relevant to this subject area, such as evaluations in the edge-cloud continuum and architectures of state-of-the-art serverless computing frameworks. It not only shows other recommendation approaches but also introduces other related work in the context of this thesis. The system model our framework is based on together, with its workflow and custom schedulers, is described in Chapter 4. Chapter 5 details the approach and methodology of our recommendation and evaluation of the results. The discussion of the results and the recommendation of orchestration architectures from different perspectives on different scenarios are presented in Chapter 6. Lastly, Chapter 7 concludes this thesis, answers our formulated research questions, and outlines future work.

CHAPTER 2

Background

This section presents the fundamental concepts of edge intelligence, edge-cloud continuum, serverless computing, and different orchestration architectures in the literature. Furthermore, information about the tools used in the experiments is also presented.

2.1 Edge Intelligence in the Edge-Cloud Continuum

Cloud computing is a paradigm that offers on-demand computing services through a pool of computing resources. Among other things, storage and computing resources are services provided by the cloud [27]. These cloud data centers have the performance requirements for hosting AI models. However, they are often far away from the users [43]. This distance results in higher latency, which is not feasible for current training and inference tasks in AI applications. Furthermore, moving the resources closer to the users mitigates the bandwidth pressure caused by the massive amount of generated data [57].

Therefore, edge computing has been introduced, allowing applications to be executed closer to the users and thereby reduce latency compared to cloud computing [6]. Within the edge computing paradigm, applications and services are handled at the edge of a network instead of sending them to the cloud. One other unique characteristic, besides low latency, is heterogeneity. In edge computing, heterogeneity refers to the various technologies, architectures, and infrastructures used by edge computing elements. Significantly, the variations in software and hardware technologies in the end devices are the main factors of the heterogeneity [27].

Edge computing alone cannot match cloud computing's computational power and resource capacity. Combining these two technologies allows edge computing to reach its fullest potential. This cloud and edge computing combination is also called the edge-cloud continuum [48]. Moreover, edge intelligence applications operate on multiple layers, so the two paradigms must be considered together [74].

Edge intelligence is a technology where edge computing and AI are integrated together [57]. There has yet to be a clear definition [74, 14] for edge intelligence. However, current literature agrees that it at least refers to executing an AI model on the edge [57]. Using the edge enables very low latency and context-aware AI applications, which can handle massive amounts of data from heterogeneous data sources [57]. Edge intelligence can be further classified into AI for edge and AI on edge. The former describes a research direction where AI technologies are used to solve constrained optimization problems in edge computing. The latter researches how to run AI models on the edge while adhering to QoS standards [14].

There are multiple possible use cases for edge intelligence in different domains, such as smart agriculture, smart cities, and more. Raith and Dustdar [57] present several use cases for edge intelligence applications. They describe how a hypothetical company selling agricultural products could use edge intelligence to improve its business. To name one use case, the company could use AI that estimates soil fertility to help farmers with optimal seeding.

In edge intelligence applications, inference is one of the main operational tasks. While it benefits from the edge in the form of lower latency, bandwidth reduction, and more, the task also suffers from one of the drawbacks of edge computing: resource-constrained devices and the resulting performance issues. The inference task is susceptible to latency and performance issues. Therefore, it is vital to provide sophisticated orchestration to handle the heterogeneity found in edge computing [57].

Since edge computing consists of multiple heterogeneous elements and technologies, resource management has proven challenging. Some resource management solutions exist, but most are unsuitable for highly distributed and heterogeneous environments [69]. The heterogeneity can also be observed in the scenarios of [60]. Based on real-world use cases, the authors demonstrate that edge computing applications consist of different network technologies and heterogeneous computing devices. They introduce multiple infrastructure scenarios and describe the different hardware used in them. The hardware also differs vastly in computational power between these scenarios. Some use computation-heavy hardware, such as edge data centers or compute clusters, while others only use single-board computers and mobile devices. Furthermore, different network capacities can be observed between these scenarios as well. This heterogeneity makes operating data-intensive applications challenging. In order to deal with the complexity of such systems, serverless computing has proven itself an increasingly compelling option[62].

2.2 Serverless Edge Computing

Serverless computing, also called Function as a Service (FaaS), is a service model where the user only needs to provide the application in the form of stateless functions. The provider manages other service responsibilities, such as resource allocation, load balancing, and more. Serverless computing follows an event-driven architecture. Providers define a set of event sources that can trigger user-defined functions. Some of the possible events are, for example, an HTTP request from a user interface or a change to a database. Users can then define rules and bindings to event sources to control when a function should be executed [44]. FaaS can be used in different application domains. While the most familiar domain for serverless computing is the web service domain, serverless computing has also found its way into Big Data Analytics, Internet of Things, Machine Learning, and more. Although each domain still faces challenges with serverless computing, its potential is evident [9].

Serverless platforms come with distinctive attributes that, among other things, include auto-scaling capabilities. This feature enables automatic scaling of resources to meet demand. It should also be able to scale to zero when there is no traffic. Therefore, billing can be implemented, where users are only charged when functions are executed, and nothing is charged when functions are scaled to zero. Another characteristic of serverless platforms is that many already support function code written in multiple languages, including Java, Python, JavaScript, and more. Furthermore, providers set limits to runtime requirements of function code, such as maximum memory, maximum execution duration, and more, to preserve the system's flexibility [44].

As mentioned earlier, some edge computing challenges are also related to serverless computing. The most significant challenge is to improve overall performance since edge devices are resource-constrained. Therefore, serverless platforms must keep overhead to a minimum to avoid performance degradation. Another strategy to handle resource constraints is offloading, where workloads that cannot be executed locally are dispatched to other devices on the same or higher layer. Another challenge is latency because it is vital to keep communication between devices as fast as possible, especially for latency-critical applications. In similar scenarios, cold starts can also create bottlenecks. A cold start occurs when a function is executed for the first time. An overhead is created in this case due to the need to initialize the container and the underlying services [9]. Furthermore, function placement is NP-hard, making it a challenging problem to solve [55]. To mention edge intelligence-related challenges, dealing with latency and bandwidth consumption in model executions and providing an optimal splitting of inference tasks across the edge-cloud continuum proves difficult. These challenges are attributable to the heterogeneity of the required AI accelerators for the various tasks and environments [48]. Furthermore, there has yet to be a clear indication of which orchestration architecture performs best. Current literature supports multiple approaches and architectures, see Section 3.2, indicating that the best approach is yet to be found. Therefore, this thesis examines the performance and impact of centralized, decentralized, and distributed orchestration architectures.

2.3 Orchestration Architectures

This section briefly introduces the main orchestration architecture strategies named centralized, distributed, and decentralized [30]. A visual illustration of these architectures can be seen in Figure 2.1. Furthermore, it describes the primary responsibilities of

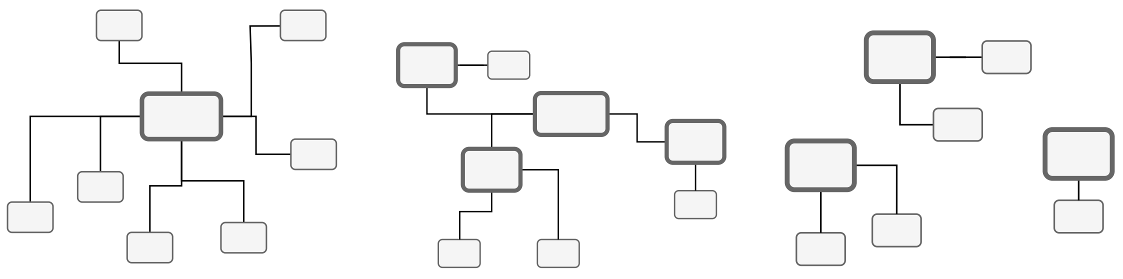

(a) (b) (c)

Figure 2.1: Orchestration architectures. (a) Centralized; (b) Decentralized; (c) Distributed. Adapted from [13]

orchestration management.

2.3.1 Architectures

Centralized

Centralized approaches can be compared to a master-worker model, where a single control component exists with a complete overview of the distributed system. Moreover, only one controller component makes decisions regarding task- and resource allocation between nodes. See Figure 2.1a for a figurative representation of the architecture. On the one hand, a centralized approach makes it easier to evolve toward upcoming technologies and maintain the system. On the other hand, through the single point of failure that the central component represents, scalability and reliability suffer [45, 13, 30].

Decentralized

Like the centralized approach, decentralized architectures also contain a global controller component. However, in contrast, local controller components that manage a group of nodes also exist. In other words, intra-cluster decisions are made on a central controller, whereas inter-cluster decisions are made on the local cluster controller. This strategy is depicted in Figure 2.1b. Although this approach performs better than a centralized approach regarding scalability and reliability, it needs to improve in maintenance and evolution toward upcoming technologies [45, 13, 30].

Distributed

In a distributed approach, there is no master node. Computing clusters, even when consisting of a single node, can make decisions autonomously. The nodes/clusters can sometimes communicate and exchange local information with each other. While distributed architectures provide reliability and scalability, they suffer at the expense of expanding to upcoming technologies and maintaining the system [45, 13, 30]. During this thesis, we define distributed systems as architectures with computing clusters that do not communicate with one another. Figure 2.1c represents a figurative depiction of the architecture.

2.3.2 Orchestration management responsibilities

As mentioned above, orchestration management is a critical task of serverless edge computing. The aim of this orchestration layer should be able to predict on-demand resources and provide an efficient scheduling strategy. Most serverless systems use a load-balancer, autoscaler, scheduler, and resource monitor to handle the orchestration [58]. While the load-balancer manages the resource usage to avoid overloading a single resource, the resource monitor collects resource utilization information and communicates it to the control components. In our case, load-balancer, autoscaler, and scheduler are included in the controller, which allows the controller to resolve scheduling challenges, among others, on resource-level and instance-level [37]. These challenges are not to be underestimated since scheduling serverless functions on a serverless platform based on user Service-level Agreement is an NP-hard problem [44]. Additionally, the performance of the individual control components also contributes significantly to the system's performance.

Regarding resource provisioning, CPU and RAM usage are typical values that the controller considers. The controller allocates resources when a new instance requests them. One of the critical challenges of resource provisioning is to provide the *perfect* amount of resources. In order to avoid over-provisioning, controllers can often dynamically adjust resources with the help of the real-time resource monitor. However, it is also possible to use Deep Reinforcement Learning or other AI technologies to further aid the controller with resource provision decisions [37].

On instance level, the load-balancer takes control of the routing. It is designed as a router, which helps in routing requests between nodes while keeping the load as equal as possible between the nodes. The main approaches can be classified into hash-based or multi-objective-based. The hash-based strategy is the more basic one, where a hash function is used for each function, deciding its routing. If the target is unavailable, it looks step-wise for the next node. In contrast, multi-objective-based strategies for selecting the target node are based on multiple factors such as response time, resource utilization, and more [37].

Furthermore, not only scheduling, but also scaling plays a vital role in the orchestration layer. Scaling refers to automatically scaling resources to meet varying resource demands from functions. In order to reach high resource efficiency, resources are expected to scale out when there is a rise in demand and scale in when the demand diminishes. While there are multiple types of scaling, we only describe horizontal scaling since this mechanism is used in our framework. Horizontal scaling refers to the capability to provide and remove new machines to meet the current demand [44]. In our case, we scale function instances as Kubernetes Pods, described in Section 2.4.1.

Ultimately, these resource mechanisms are used to reach, among others, high resource utilization, high availability, high scheduling throughput, and application-specific QoS

[64]. As already mentioned, we utilize Kubernetes to implement our orchestration layer. Since multiple well-known open source frameworks, such as Knative [29], OpenWhisk [50], and OpenFaas [39], also utilize Kubernetes, we can assume Kubernetes to be a solid choice.

2.4 Tools and Technologies

The following sections introduce tools and technologies used in the experiments.

2.4.1 Kubernetes

The content of this section is based on the official documentation of Kubernetes [34].

Kubernetes is a container orchestration platform that was open-sourced by Google in 2014. Due to its popularity, a lot of tools and extensions exist. It follows a declarative style, where users describe the desired status of an object within the system, and Kubernetes tries to match the current state to the desired state. Furthermore, Kubernetes uses container deployment as its deployment strategy. One of the benefits of containers is their lightweight nature, unlike, for example, virtual machines. As mentioned earlier, edge computing elements are constrained in resources. Therefore, using a lightweight deployment is beneficial, enabling the edge elements to reserve more resources for computational tasks.

Furthermore, Kubernetes comes with many features that facilitate the deployment of applications. It enables users to run a distributed system resiliently by providing features such as self-healing containers, service discovery and load-balancing, automated rollouts and rollbacks, and more.

The following subsections cover some components of interest for the thesis. Figure 2.2 depicts the architecture of Kubernetes.

Cluster basic elements

In simple terms, when deploying Kubernetes, a cluster is created. Each cluster consists of worker machines that can run the applications.

- **Nodes**
  In Kubernetes, these worker machines are called nodes. Depending on the circumstances, these Nodes can be physical or virtual machines controlled by the control plane. In order to run the workload, Kubernetes places containers (mostly Docker containers [15]) into Pods, which runs on Nodes.

- **Pods**
  These nods host Pod(s), the computing units that run the application. A Pod is a group of one or more containers with shared storage and network resources. Pods

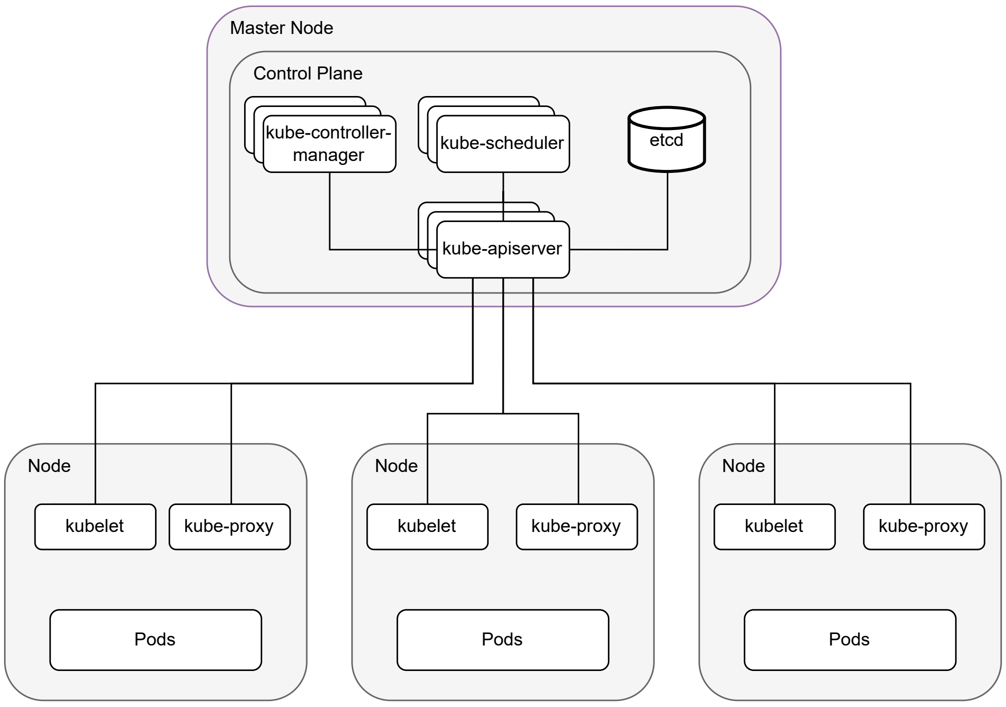


Figure 2.2: Kubernetes architecture

usually have a specification on how to run the containers and are always co-located and co-scheduled on the same node in a shared context. The most common use case of Pods is to run a single container and use the Pod as a wrapper for the container since Kubernetes manages Pods but not the containers directly.

Control plane components

The control plane is the mind of the cluster, which makes all global decisions. The control plane consists of four different components, described as follows.

- **kube-apiserver**
  The kube-apiserver component exposes the REST Kubernetes API. It behaves like the front end of the control plane and enables users access to the cluster's shared state, with which all other components also interact. Moreover, it configures and validates data for objects such as pods, nodes, controllers, and more.

- **etcd**
  Etcd is a consistent and highly available key-value store system. Kubernetes uses etcd to store all cluster data, such as configurations, current cluster state, metadata, and more.

- **kube-scheduler**
  The kube-scheduler's responsibility lies in assigning newly created pods to nodes when no assigned node is stated in the pod configuration. The selection of a node is based on multiple factors, which can also be controlled by the user, such as data resource requirements, policy constraints, affinity and anti-affinity specifications, and more. Furthermore, Kubernetes allows users to write custom schedulers as well. Since the default implemented scheduler from Kubernetes cannot fully meet the requirements of edge applications [63], we decided to implement a custom scheduler for our experiments.

- **kube-controller-manager**
  This component runs the controller processes of the cluster. These processes are non-terminating loops that regulate the cluster. Each controller has its control loop, which observes at least one resource type. They watch the current state of the cluster through the API server and, if needed, request changes to get the current state closer to the desired state. These request changes are sent to the API server, which then initiates the needed actions to execute the wanted changes inside the cluster.

Node Components

The following components run on every node. Their function is to maintain running pods and provide the runtime environment.

- **kubelet**
  Each node in the cluster has a kubelet agent running. Its purpose is to manage containers and ensure they are created and running within given specifications.

- **kube-proxy**
  Similar to kubelet, each node runs a kube-proxy. It is a network proxy that enables network communication to the pods from inside and outside.

2.4.2 Ansible

The content of this section is based on the official documentation [2].

Ansible was created to reduce the complexity of system configuration, deploying software, and more on multiple computers. It is an IT automation tool with simplicity and ease of use as its main goals and a focus on security and reliability. For transport, OpenSSH [49] is used, which significantly reduces the security exposure. Furthermore, it is designed to be audible to humans, even if they are unfamiliar with the program.

In order to use Ansible, the Ansible CLI tools must be installed on a machine, which is called the control node. Afterward, the IP addresses of the target devices or hosts need to be set. The basic units of Ansible execution are called playbooks. These playbooks are written in YAML syntax, which consists of tasks to be executed on the target hosts. Playbooks are further divided into plays, roles, tasks, and handlers. A play is the primary context of an execution and maps the hosts to the tasks. Inside a play, roles can be used, which consist of a limited distribution of reusable Ansible content, such as tasks, handles, files, and templates. A task defines an action to be applied to a host. Handlers are tasks that can only be triggered by a previous task.

2.4.3 Galileo

We extend the framework created by Raith et al. [59]. This framework was created to provide a tool for evaluating resource management strategies for edge-cloud clusters. It is based on Galileo [61] and was, among other things, extended to Kubernetes with additional system monitoring instruments. The end product of the extension created by Raith et al. allows for conducting scalable and reproducible experiments and fine-grained monitoring on Kubernetes clusters.

The primary analytics data from the framework can be divided into two categories: traces and resource usage. The resource usage is captured by their fine-grained monitoring tool *telemd*. Its task is collecting monitoring data from Kubernetes pods, such as CPU, I/O, network, and more. Traces, however, hold much information about the HTTP requests, such as round-trip time, latency, execution time, execution location, and more.

The experimental workflow of the framework is divided into three phases: pre-experimental, runtime, and post-experimental. The framework handles the runtime phase entirely, while user input is necessary for the other two phases. For the pre-experimental phase, users must provide an application, the cluster nodes, clients creating HTTP requests, and a workload. Any containerized function works for the application as long as it exposes an HTTP endpoint. The framework is responsible for the runtime of experiments, including the setup of control components, spawning applications, configuring clients, and managing the recording of telemetry and trace data. Afterward, for the post-experimental phase, the framework provides a gateway for Jupyter Notebooks, allowing users to access recorded data during the experiments easily.

CHAPTER 3

State of the Art

This chapter provides an overview of recommendation surveys, similar work regarding evaluating aspects of serverless computing, and orchestration architectures in state-of-the-art frameworks.

3.1 Recommendation Systems and Surveys

During the literature research, only one paper [73] was found that shared a similar idea to this thesis. The authors created a simple recommendation system, which facilitates users in selecting an appropriate hardware accelerator for their edge applications. Unlike us, they based their recommendation only on latency, power consumption and a cost evaluation. Furthermore, they conducted the experiments on different hardware, and used multiple machine learning inference tasks as their application for their benchmarks. As there is limited literature regarding recommendation systems for edge computing, this section also covers surveys of recommendation systems for cloud computing to showcase the many different approaches for a recommendation system. The two following studies analyzed existing cloud service recommendation systems and their approaches, exposed issues with current systems, and proposed future research directions.

Sun et al. [66] surveyed state-of-the-art cloud service selection approaches. The analysis is based on multiple aspects, such as selection techniques, data representation models, considered characteristics, and more. Furthermore, they provide a more in-depth analysis of different service selection approaches: optimization-driven, logic-driven, and multi-criteria-based decision-making. The authors also identified eight open issues regarding cloud service selection approaches.

Aznoli and Navimipour [4] conducted a review of existing techniques in cloud recommender systems. The techniques were divided into four categories: collaborative filtering, demographic-based, knowledge-based, and hybrid. Furthermore, the authors highlighted the advantages and disadvantages of each mechanism and compared and reviewed the strategies in terms of scalability, availability, and more.

3.2 Architecture

To showcase that current literature comprises multiple orchestration architectures and approaches, this section provides an overview of different state-of-the-art orchestration frameworks and their architectures.

3.2.1 Centralized

Pfandzelter and Bermbach[54] propose *tinyFaaS*, an innovative lightweight FaaS system specifically designed to run on low-performance edge nodes. Moreover, it is designed to run on a single node. It follows a centralized architecture style where a management service creates functions within the platform. When a new function is requested, the service orchestrates all necessary steps to add it to the platform. Furthermore, it is possible to delete and modify functions at runtime.

Wang et al. [68] developed *LaSS*, a platform focusing on latency-sensitive serverless computations on the edge. Its architecture is based on the architecture of Apache OpenWhisk [50]. The authors redesigned the controller of the original decentralized approach from OpenWhisk to a centralized approach. Moreover, they implemented a profound allocation strategy, in which, during the presence of overload, each function has a guaranteed minimum of resources and resource reclamation methods, in which resources from over-provisioned functions are reassigned to under-provisioned ones.

Xiong et al. [70] created *KubeEdge*, an infrastructure for the edge computing environment to extend the cloud capabilities to the edge based on Kubernetes. The architecture consists, among others, of an edge controller, which can be integrated into Kubernetes as a controller plugin. This controller aims to manage edge nodes and the cloud as one logical cluster, allowing *KubeEdge* to schedule, deploy, and manage container applications across the cloud and edge. The authors also introduce *KubeBus*, which connects the edge and cloud into one virtual network, and a *MetadataSyncService*, which synchronizes metadata between cloud and edge.

3.2.2 Decentralized

Yu et al. [71] propose a scalable and low-latency serverless platform called *Pheromone*, which follows data-centric function orchestration. Regarding the architecture, the authors propose a decentralized, two-tier scheduling approach. Each request first gets processed by the global coordinator, which handles a disjoint set of workflows and routes the request to a local scheduler at a node. The decision is based on node-level information reported by local schedulers. The global coordinator aims to forward requests from overloaded to free nodes and to drive the execution of a large workflow running on multiple nodes. The local scheduler tries to keep data locality as high as possible and makes its decisions based on the information given by the function executors. Furthermore, it only routes function requests to idle executioners to avoid concurrent invocations and resource conflicts.

Li et al. [36] propose a scalable, flexible, and high-performance serverless platform named *funcX*, essentially serving, managing, and distributing tasks as a broker. It follows a decentralized structure with a global management service as an entry point and locally deployed *funcX* endpoints. The service provides, among others, monitoring features, function registration, and function output handling. Function requests from users are registered by the service and then forwarded to a suitable endpoint by a forwarder. These endpoints have multiple components that handle scheduling, load balancing, resource allocation tasks, and function execution. The function's output is then delivered to the service and made available to the user.

Moritz et al. [47] present a decentralized cluster-computing framework called *Ray* to handle the new requirements of emerging AI applications. Its architecture consists of two layers: the application layer and the system layer. The application layer contains an API and its computation model, while the system layer handles task scheduling and data management. In the system layer lays, among others, their global control state, which maintains the entire control state of the system and enables all system components to be stateless. Furthermore, the authors claim that every component in the system layer is fully fault-tolerant and horizontally scalable. The authors developed a bottom-up approach for task scheduling. Firstly, tasks created at a node are submitted to its local scheduler. However, if the node is overloaded or cannot satisfy a task's resource requirement, the task is forwarded to the global scheduler to find a more suitable node.

3.2.3 Distributed

Baresi et al. [6] created a serverless-based framework called *NEPTUNE* for large-scale edge applications. It can allocate CPU resources dynamically, use GPU power, balance the workload on multiple nodes, and place functions considering the user's location. A three-level control hierarchy is introduced to manage the functions. Each controller works independently of the others, and controllers do not interact on the same level. The highest layer employs a single controller, which splits the topology into non-overlapping, independent communities. At the next level, another controller is used for each community to manage routing, horizontal scaling, and GPU/CPU utilization. Lastly, at the lowest level, each function instance is managed by a controller handling vertical scaling to ensure that the set response times are met.

The design from *NEPTUNE* is inspired by the work of Baresi and Quattrocchi [7] called *PAPS*, a framework for large-scale edge applications. The framework follows a hierarchical architecture consisting of three control layers. The first layer partitions the topology into potentially overlapping and delay-aware communities. The second layer operates at the community level and controls horizontal resource allocation and function placement. Lastly, the third layer works at the node level and is in charge of vertical scaling to minimize Service Level Agreement violations.

Ciavotta et al. [12] propose a serverless-based and distributed architecture called *DFaaS* to balance the traffic load across edge nodes. In the suggested architecture, each edge node consists of three components: Agents, Proxy, and FaaS platform. Every agent is responsible, among others, for creating and managing a peer-to-peer network used for discovering other agents and communicating topology updates, monitoring different metrics, and configuring the proxy. Proxies receive incoming load information and the current load distribution from the local agent. They use this information to partition the load among the local FaaS platform and others on different nodes. Lastly, the FaaS platform instantiates and executes the functions.

3.3 Evaluations

Since the recommendation is based on KPIs, this section presents evaluations in the edge cloud continuum.

Palade et al. [51] evaluated the response time, throughput, and success rate of functions deployed on Kubeless [31], Apache OpenWhisk, OpenFaaS [39] and Knative [29]. The JMeter framework collects the metrics, and the serverless platforms were installed on a bare-metal, single-master Kubernetes cluster. While we only focus on quantitative metrics, Palade et al. also analyzed qualitative metrics such as programming language support, open source license, ease of deployment, and more. In contrast to us, their framework entirely depends on the control components of Kubernetes.

Mohanty et al. [46] provided a feature comparison of the open-source serverless frameworks Kubless, OpenFaaS, Fission [20], and OpenWhisk. Furthermore, the authors evaluated the performance of Kubless, Fission, and OpenWhisk deployed on a Kubernetes cluster. In contrast to us, the authors assessed the performance under different loads. However, they only analyze the response time and ratio of successfully received responses.

Lynn et al. [42] provided an overview and multi-level feature analysis of seven enterprise serverless computing platforms for the enterprise environment. They collected their information from vendor websites, platform documentation, and websites from specialists comparing these platforms.

Lloyd et al. [38] investigated how functions deployed on AWS Lambda and Azure functions influence the performance of microservices. While our focus lies more in quantitative metrics such as round-trip-time or CPU usage, Lloyd et al. focused on identifying factors influencing the performance of function microservices for serverless computing platforms. Some of the factors are load-balancing and infrastructure elasticity. While we used a local Kubernetes cluster, the authors used AWS lambdas and Azure functions for their experiments.

By invoking concurrent functions, Lee et al. [35] evaluated the CPU performance, network bandwidth, and I/O throughput of public enterprise serverless environments. They analyze fewer quantitative metrics than us, but unlike us, they also analyze qualitative metrics, such as language support and feature comparison. Furthermore, we only focused on serverless computing, while they also showcased the difference between serverless computing and virtual machines regarding cost efficiency and resource utilization. Additionally, all our experiments were conducted on a local Kubernetes cluster, while Lee et al. used various serverless enterprise environments such as AWS Lambda and Azure Functions.

Böhm and Wirtz [8] presented a quantitative analysis of three different edge orchestration strategies, named Capillary Container Orchestrator [67], Boundless Resource Orchestrator [72], and Enhanced Container Scheduler [22]. Furthermore, they provided domain-specific quantitative metrics, which can be used to evaluate edge orchestration strategies. Unlike us, all experiments were conducted on the same infrastructure, consisting of a three-layer architecture following a heterogeneous device structure. The authors deployed a CPU-intensive application to evaluate the strategies based on their metrics. Like us, their aim is to create a recommendation system that should be able to create a suggestion per an agreed QoS.

CHAPTER 4

Framework

This chapter describes the extension of the Galileo framework used for collecting metric data and the custom-made schedulers. This thesis requires multiple experiments, so creating an easy-to-use experiment setup is vital. The complete framework consists of four main steps, where many tasks are automated, which ensures a rapid setup. After experimenting, the collected data needs to be preprocessed and analyzed. We support a wide range of metrics requiring a different degree of preprocessing before they can be used. The first section of this chapter describes the system model we use for our framework. The next section gives more information about the steps of the setup flow in Figure 4.1. The last section describes the developed schedulers for the centralized, decentralized, and distributed architecture.

4.1 System Model

The framework manages a set of clusters $C = \{Cluster_1, Cluster_2, ..., C\}$, where each cluster consists of a set of nodes $N = \{Node_1, Node_2, ..., N\}$, for each $Cluster_i \in C$. Since it should be possible to define multiple resource capabilities and different areas of responsibility for the nodes, each node is represented as a tuple $Node = (role, cores, ram, initialcpu)$. In this representation, $role$ represents a node role, $cores$ represents an amount of CPU cores, ram represents a RAM amount, and $initialcpu$ represents a value for initial CPU usage. Furthermore, the zone name for cluster c is described as n_c^{zone}. This attribute can be helpful as zone names can be used as a grouping mechanism, for example, dividing a large cluster into smaller clusters based on similar characteristics such as location. As a result, tasks can be allocated more efficiently.

The framework oversees a set of function replicas F with $f \in F$. The location of a function replica f is denoted as f_n, implicating that the function runs on the node n. Each function is further described with a CPU requirement f_{CPU}^{req} in milliCPU and RAM requirement f_{RAM}^{req} in megabytes. MilliCpu represents the CPU time a function requests

[32]. Furthermore, we introduce t_f^{rtt}, which describes a function's round-trip time target duration f.

Moreover, the framework handles a set of clients U, where a client $u \in U$ generates requests. $R_{u,n}$ represents the set of requests from client u sent from node n.

Furthermore, each node is connected through an inter-network. The framework supports the possibility of emulating network latency between nodes, where we define $l_{n1,n2}$ as the latency value between node n1 and node n2.

As for the orchestration components, firstly, we define LB as the set of load-balancers with $lb \in LB$. Then we introduce $rr_{r,f}$, which represents to which function replica f a request r is routed.
Then we define AS as the set of autoscalers with $as \in AS$, and $dr_{c,rtt}$ as the desired amount of function replicas for cluster c based on the round-trip time target duration rtt. Then, we define $SCHED$ as the set of schedulers with $sched \in SCHED$ and $p_{f,n}$ as the placement of a function replica f in node n. Table 4.1 lists all symbols.

Variable	Description
C	Set of clusters
N	Set of nodes
n_c^{zone}	Zone name of cluster c
F	Set of function replicas
f_n	function replica on node n
f_{CPU}^{req}	CPU requirement of function in milliCPUf
f_{RAM}^{req}	RAM requirement of function in megabytes f
t_f^{rtt}	Target round-trip time of function f
U	Set of clients
$R_{u,n}$	Set of requests from user u from node n
$l_{n1,n2}$	Network latency between node $n1$ and node $n2$
LB	Set of load-balancers
$rr_{r,f}$	Function replica routing target f for client request r
AS	Set of autoscalers
$dr_{c,rtt}$	The desired amount of function replicas for cluster c, based on round trip time rtt
$SCHED$	Set of schedulers
$p_{f,n}$	The placement of a function replica f on node n

Table 4.1: Symbols

4.2 Experimental Setup

In order to conduct the different experiment settings, we created an edge-cloud VM-based provisioning system that uses the Galileo framework [59], with some extensions. Figure 4.2 depicts all components of the provisioning system. The complete framework can be broken down into four significant steps, described as follows. An overview of the structure is illustrated in Figure 4.1. In this figure, parts marked as legacy are implementations we can use directly from Galileo. The steps marked as partially extended/new describe steps in which we either extend existing implementations or, in addition to this, incorporate our custom implementations in the step. Steps marked as new are parts where we cannot use existing implementations of Galileo and, therefore, include our custom implementations.

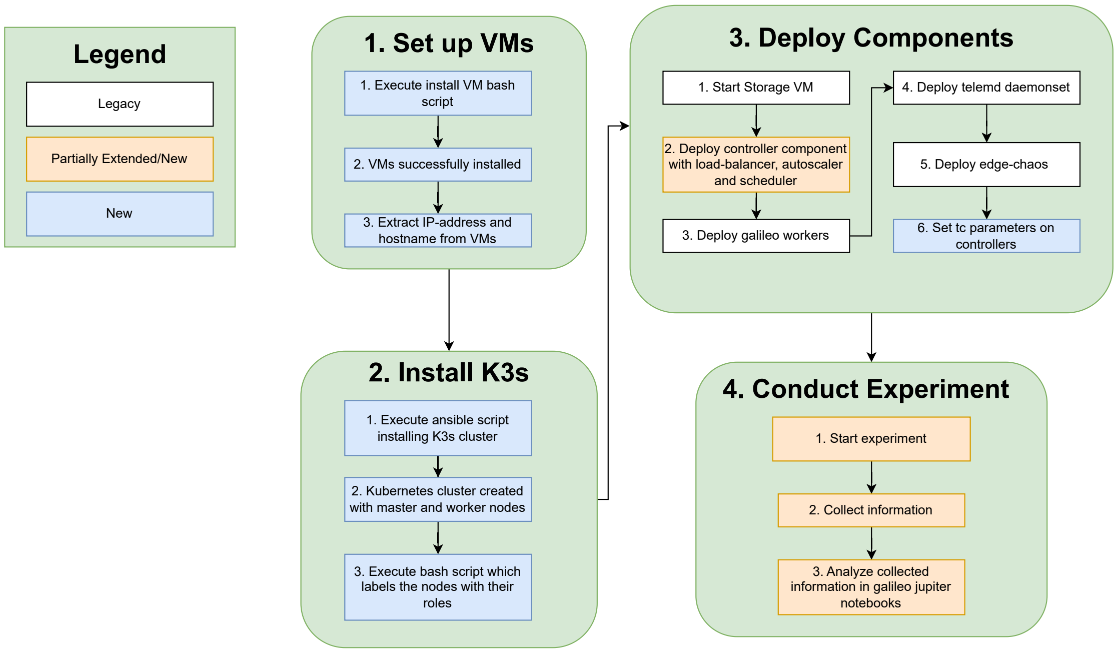

Legend
Legacy
Partially Extended/New
New
1. Set up VMs
1. Execute install VM bash script
2. VMs successfully installed
3. Extract IP-address and hostname from VMs
2. Install K3s
1. Execute ansible script installing K3s cluster
2. Kubernetes cluster created with master and worker nodes
3. Execute bash script which labels the nodes with their roles
3. Deploy Components
1. Start Storage VM
2. Deploy controller component with load-balancer, autoscaler and scheduler
3. Deploy galileo workers
4. Deploy telemd daemonset
5. Deploy edge-chaos
6. Set tc parameters on controllers
4. Conduct Experiment
1. Start experiment
2. Collect information
3. Analyze collected information in galileo jupiter notebooks

Figure 4.1: The setup flow of an experiment using the framework

1. Set up VMs

As different infrastructures are tested, the creation of VMs must be simple and automated. Since VM creation is not part of the Galileo framework, we provide different bash scripts, which can create and tear down our infrastructures. The script enables the user to control the amount and resource configuration for each VM creation and is easily configurable. In other words, it creates C and N from Table 4.1.

To facilitate the cluster configuration in the following steps, we recommend that the created VMs follow a specific naming scheme that includes cluster roles and information in its name. The bash install script also controls this configuration. This script makes creating fully customizable clusters with different node types and zones possible. This corresponds to n_c^{zone} and $roles, cores, ram$ from the $Node$ tuple. Furthermore, we provide a script that extracts the IP addresses of the newly created VMs, which is vital information needed for the next step.

2. Install K3s

After the VMs are up and running, the next step is to create the cluster. Like Step 1, installing K3s is not part of the original Galileo framework. Since we want to be as lightweight as possible, we decided to go with K3s [25], a lightweight Kubernetes built for IoT and Edge. For the actual install process, instead of installing and configuring all the requirements manually, we use the official Ansible playbook [26] provided by K3s to install K3s. Configuring the host file and some configuration variables in the playbook allows it to create a K3s cluster fully automatically. Usually, using Ansible requires manually creating the host configuration by typing in the IP addresses of the cluster nodes. In order to overcome this shortcoming, we provide a bash script that automatically fetches the IP addresses of the newly created VMs and creates the host configuration file. Since we use Ansible, users can effortlessly add custom tasks after cluster initialization to prepare the cluster for further experiments.

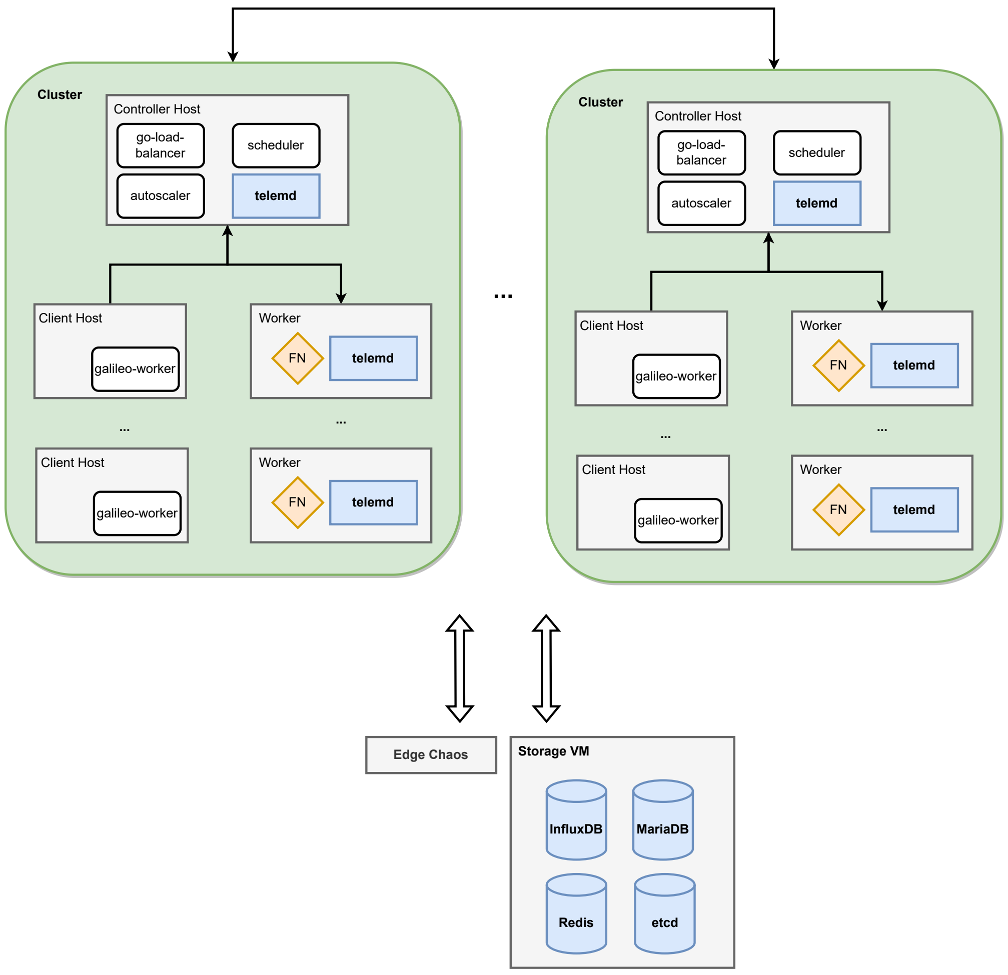


Figure 4.2: All components of the provisioning system

3. Deploy Components

Figure 4.2 illustrates the system after deploying all components. The following paragraphs introduce the components of Figure 4.2, and their responsibilities.

After the clusters are set, it is time to start a storage VM, where the storage systems run, including a Redis, InfluxDB, and MariaDB instance. These three storage systems are used for event publishing and metric processing. Furthermore, we start an etcd instance, which is used for our load-balancer. This storage system is a prerequisite for the Galileo framework and needs to be done manually. Afterward, deployment of custom orchestration and the Galileo components can start.

Firstly, the controller components such as autoscaler (AS), scheduler($SCHED$), and load-balancer(LB) are deployed, where the autoscaler and load-balancer are provided by the Galileo framework. The autoscaler [16] and scheduler [16] are written in Python, and the load-balancer in GO. They create and manage the set of function replicas F.

The responsibility of the autoscaler is to manage the dynamic allocation of function replicas during the experiment. It is based on the official default Kubernetes HPA [33] and uses round trip time to determine the number of replicas. Equation (4.1) is used for calculating the number of desired replicas and relates to $dr_{c,rtt}$.

$$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)] \tag{4.1}$$

The currentMetricValue correspondents to the 50th percentile of round trip time (t_f^{rtt}) values of user requests sent and completed in the last ten seconds. The desiredMetricValue corresponds to a chosen target duration (see Table 4.2 and Table 5.4). Furthermore, the scale-down is inspired by OpenFaaS [39], where scale-down happens gradually. In this case, the largest possible scale-down is 20 pods for each scale-down decision. Moreover, if there are no completed user requests in the last seconds, the autoscaler also scales down 30% of the currently running pods. Furthermore, there is a controllable reconcile interval integrated into the autoscaler. More information about this parameter can be found in Table 4.2.

Scheduling ($p_{f,n}$) new replicas and therefore defining the location of a replica (f_n) is the responsibility of the scheduler component. The Galileo Framework provided no scheduler. Therefore, we developed custom schedulers for each orchestration architecture. In general, all schedulers work similarly in the sense that the most optimal node is selected based on the CPU utilization. For a more detailed overview of the schedulers, see Section 4.3.

The earlier mentioned etcd store provides up-to-date weight values for the Go-based load-balancer [17]. The load-balancer follows a weighted round-robin strategy, where the weights are based on latency. Its task is to handle the requests from galileo workers by forwarding the calls to Pods following the weight distribution published in etcd. The forwarding of requests refers to $rr_{r,f}$

The next step is to deploy the galileo workers, which represent the clients U. They act like users and generate the workload. The Galileo framework allows multiple workload profiles as well. Finished requests with information such as latency, round-trip time, source, and target cluster are published to Redis. This request refers to $R_{u,n}$ and can help further understand how the application and cluster perform.

Then the telemd [18] deamonsets are deployed. They collect the resource usage of each node and publish them to Redis. This data can be used to get more insights into how a cluster handles the workload.

Afterward, edge-chaos [19] is started. Edge-chaos allows us to create CPU stress, which is needed for the weak worker nodes (see Section 5.1.1). It is used for the *initialcpu* value from the *Node* tuple.

Lastly, the network latency and available bandwidth between the compute units in the system gets emulated using the Linux network traffic shaping tool tc, representing $l_{n1,n2}$. This integration is also part of our extension of the Galileo framework, and the latency and bandwidth values are fully customizable.

We also provide a bash script to teardown all mentioned components, which is helpful when multiple experiments are conducted consecutively.

4. Conduct Experiment

All required components are up at this point, and it is possible to start an experiment. For an experiment, a set of parameters is supported. To get closer to real-world applications and depict the impact of cascading delays during orchestration, we created multiple parameters, such as a delay for the scheduler, a maximum amount of replicas, and a target duration and reconciliation interval for the autoscaler. The following Table 4.2 introduces the different parameters added to the framework. We also further improved how an experiment is started so that it is possible to run multiple experiments consecutively and with different parameter sets. Furthermore, since we provide a bash script for shutting down clusters, we can run consecutive experiments on multiple cluster setups. Unfortunately, it is impossible to run multiple experiments simultaneously since the original Galileo framework does not support this.

Table 4.2: Parameter values

Name	Description	Unit
delay	This value describes how long the scheduling decision is artificially delayed.	s
reconcile interval	This value describes at what intervals the autoscaler calculates if an up or down scale is necessary.	s
max scale	This value describes the maximum amount of replicas the autoscalers are allowed to scale up globally.	#
target duration	This value describes the target latency value used by the autoscaler to make its scaling decisions.	s

As an extension of the Galileo framework, we also provide the possibility to save the Kubernetes logs from pods to log files through a bash script. At this stage, only logs of control components are saved into log files. However, this can easily be extended to other components as well. Furthermore, it is also possible to customize where and how the log data is saved.

4.3 Custom Schedulers

This work involved the development of three different schedulers: one for the centralized orchestration, one for the decentralized orchestration, and one for the distributed orchestration. This subsection describes the developed schedulers in a more detailed way.

Centralized Scheduler

The centralized scheduler is the only scheduler in the cluster and has the ability, to get information about all nodes, regardless of their zone affiliation. The algorithm is held relatively simple. At first, the scheduler waits for a *delay* amount of seconds. Then, the scheduler searches for the node with the lowest CPU utilization, which has enough resources to host the new function instance. Algorithm 4.1 illustrates the algorithm as pseudocode. In line 6, *node is Ready* refers to a Kubernetes node status, and *node has enough resources* checks if a node has enough CPU and RAM resources to host a Kubernetes Pod. The function *get_min_cpu_usage* looks for the node with the currently lowest amount of CPU usage, and the function *bind_pod* instructs Kubernetes to run the newly created Pod on the provided Kubernetes node.

Algorithm 4.1: Centralized Scheduler

```
   Input: Kubernetes Pod in status Pending $pod$, a scalar $delay$, All Kubernetes
          Nodes capable of hosting function instances $nodes$
1  if $delay > 0$ then
2  |  $wait(delay)$
3  end
4  $ready\_nodes = \emptyset$;
5  foreach $node$ in $nodes$ do
6  |  if $node$ is Ready AND $node$ has enough resources then
7  |  |  $ready\_nodes = ready\_nodes \cup node$;
8  |  end
9  end
   // get_min_cpu_usage returns the node with the lowest CPU workload
10 $best\_node$ = get_min_cpu_usage($ready\_nodes$);
   // bind_pod binds the pod to the node
11 bind_pod($pod$, $node$);
```

Distributed Scheduler

The distributed scheduler works very similarly to the centralized one. However, there are schedulers, one for each zone, and each scheduler only manages nodes within the same zone. Therefore, in contrast to the centralized strategy, a distributed scheduler searches for the node with the lowest CPU utilization, which has enough resources to host the new function instance only inside its zone. There is no communication between the schedulers. Algorithm 4.2 illustrates the algorithm as pseudocode.

Algorithm 4.2: Distributed Scheduler

```
    Input: Kubernetes Pod in status Pending  pod, a scalar delay, Kubernetes
           Nodes capable of hosting function instances in the same zone as the
           scheduler  nodes
 1  if $delay > 0$ then
 2      wait(delay)
 3  end
 4  ready_nodes = $\emptyset$;
 5  foreach node in nodes do
 6      if node is Ready AND node has enough resources then
 7          ready_nodes = ready_nodes $\cup$ node;
 8      end
 9  end
    // get_min_cpu_usage returns the node with the lowest CPU workload
10  best_node = get_min_cpu_usage(ready_nodes);
    // bind_pod binds the pod to the node
11  bind_pod(pod, node);
```

Decentralized Scheduler

The decentralized scheduler follows a two-step approach. The first step happens in the cloud, where, in our first idea, the cluster zone with the lowest average node CPU usage is selected. However, because the cloud zone only contains strong nodes, while the other zones host strong and weak nodes with a preload of 50% CPU usage, the cloud zone's average CPU usage is significantly lower than the other zones. As a result, the global scheduler schedules all the pods to the cloud. With way more function instances on the cloud than on the other clusters, the load-balancer started to route every request to the cloud since more instances on a zone allowed for better distribution and, eventually, a lower latency. However, since we want to analyze the whole infrastructure and not just one zone, we decided to, contrary to the centralized and distributed scheduler, consider not only the CPU usage of the nodes but also the source location of the scaling request.

The pseudocode for the first step is illustrated in Algorithm 4.3. The algorithm starts with waiting for a *delay* amount of seconds. Then, similarly to the distributed scheduler, the scheduler looks for the node with the lowest CPU utilization, which has enough resources within a specific zone. In this case, we look at the origin zone of the scaling request. If there is such a node, we use the zone of this node. Otherwise, we search for the zone with the lowest CPU utilization of suitable nodes. Then, we delegate the pod to the local scheduler responsible for the found zone.

The function *cal_avg_cpu_usage_per_zone* firstly aggregates the CPU utilization of all suitable nodes for each zone. Then, it determines the average value per each zone and returns the average values and their corresponding zone.
The function *get_zone_with_min_avg_cpu_usage* uses the return value of *cal_avg_cpu_usage_per_zone* and returns the zone with the lowest average CPU utilization.

After the global scheduler rescheduled the pod, the chosen local scheduler schedules the pod to the node with the lowest CPU usage of its zone. The local scheduler in the decentralized architecture works the same as the scheduler in the distributed architecture (see Algorithm 4.2)

Algorithm 4.3: Decentralized Global Scheduler

```
Input: Kubernetes Pod in status Pending $pod$, a scalar $delay$, All Kubernetes
       Nodes capable of hosting function instances $nodes$, Map of zone name as
       key and local scheduler as value $local\_schedulers$
1  if $delay > 0$ then
2      $wait(delay)$
3  end
4  $origin\_zone = pod.origin\_zone$;
5  $ready\_nodes = \emptyset$;
6  foreach $node$ in $nodes$ do
7      if $node.zone = origin\_zone$ AND $node$ is Ready AND $node$ has enough
        resources then
8          $ready\_nodes = ready\_nodes \cup node$;
9      end
10 end
11 if $ready\_nodes$ is Empty then
12     foreach $node$ in $nodes$ do
13         if $node$ is Ready AND $node$ has enough resources then
14             $ready\_nodes = ready\_nodes \cup node$;
15         end
16     end
       // calc_avg_cpu_usage_per_zone aggregates the CPU usage of each node
          for each zone and calulcates the average value.  Retains the
          information which avg value is for which zone
17     $avg\_cpu\_usage\_per\_zone =$
        $calc\_avg\_cpu\_usage\_per\_zone(ready\_nodes)$;
       // get_zone_with_min_avg_cpu_usage returns the zone with the lowest
          average CPU usage
18     $selected\_zone =$
        $get\_zone\_with\_min\_avg\_cpu\_usage(avg\_cpu\_usage\_per\_zone)$;
19     $local\_scheduler = local\_schedulers[selected\_zone]$;
       // reschedule_pod sends the pod to the local scheduler to be
          scheduled there
20     $reschedule\_pod(pod, local\_scheduler)$;
21 end
22 else
23     $local\_scheduler = local\_schedulers[origin\_zone]$;
       // reschedule_pod sends the pod to the local scheduler to be
          scheduled there
24     $reschedule\_pod(pod, local\_scheduler)$;
25 end
```

CHAPTER 5
Evaluation

This chapter describes the evaluation procedure, visualized in Figure 5.1. Firstly, the requirements for a meaningful evaluation must be set. Therefore, the first section describes the chosen categorization and investigated metrics for the experiments. The categorization is based on real-world scenarios, and the metrics are chosen based on their description and significance found in current literature. Summarized in Step 1 in Figure 5.1. Section 5.2, Step 2 in Figure 5.1, illustrates the process of an experiment. There, we describe the setup of an experiment and the different steps from creating the experimental environment to the actual execution. There are three steps: the first two steps create the experimental environment, and the last step illustrates the actual execution of the experiment and the information processing. In order to ensure the reliability of the results, every experiment is repeated five times. The last section in this chapter, Step 3 in Figure 5.1, presents the results of the experiments.

5.1 Approach

Since the evaluation should be related to the real world, it is crucial that we first create an edge-cloud categorization based on real scenarios. This categorization helps us to build the experimental infrastructure in a way that correlates to the real world. Furthermore, to conduct an adequate evaluation, focusing on impactful performance metrics is essential. Consequently, we present various metrics, all impacting an orchestration architecture's quality of service.

5.1.1 Edge-Cloud Categorization

There are many different scenarios in which edge computing is used. Two of them are named urban sensing and Industrial IoT (IIoT) [60].

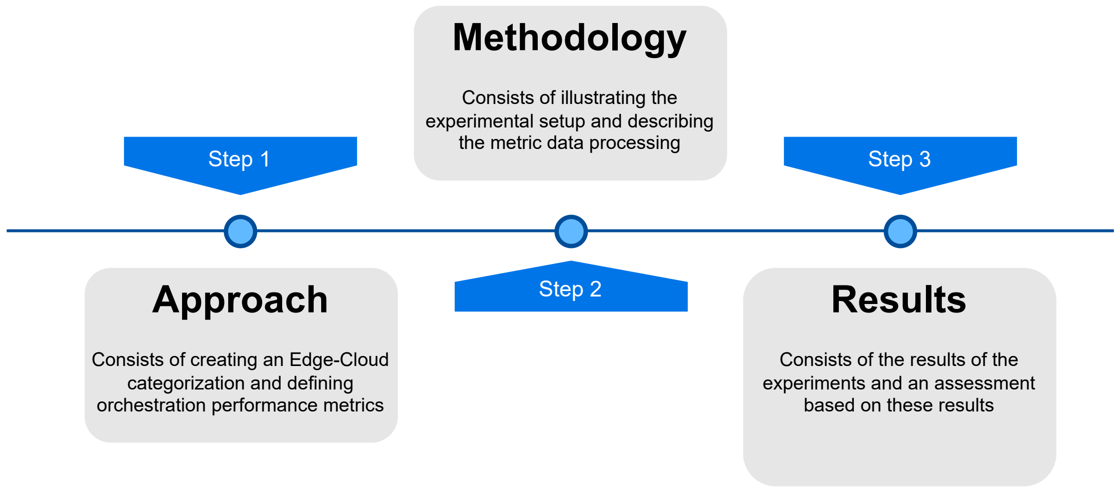

Methodology
Consists of illustrating the experimental setup and describing the metric data processing
Step 1
Step 3
Step 2
Approach
Consists of creating an Edge-Cloud categorization and defining orchestration performance metrics
Results
Consists of the results of the experiments and an assessment based on these results

Figure 5.1: Evaluation procedure

Urban sensing describes a scenario where smart cities are trying to provide environmental data for citizens and the government. One way to realize this concept is to deploy IoT nodes with cameras and sensing abilities all over the city. The collected data can then be used for monitoring applications, accident risk prediction, crowd behavior, and more. Following the Chicago project Array of Things [10], technologies such as Wi-Fi and cellular networks were used for its network infrastructure. For the nodes, Single Board Computers (SBCs) with sensors and cameras were utilized [60].

Industrial IoT is a key enabling technology for Industry 4.0 [60]. The edge computing paradigm supports IIoT in multiple situations. The edge for instance, can improve the flexibility of the network middleware. Furthermore, it enhances the interaction with generated information and enables advanced data analytics [11]. In contrast to urban sensing, IIoT uses more powerful edge computing hardware such as embedded AI hardware, small-form-factor computers, edge data centers, and more. Moreover, IIoT has a stronger network infrastructure, providing high down and upload rates.

Generally, this thesis looks at the scenarios from a higher level and only observes the different network and resource constraints. Therefore, we use a simple categorization based on high/low network and resource constraints, both elicited from the scenarios and the paper by Raith et al. [58]. This categorization allows us to emulate the heterogeneous nature of edge computing, including its different network traffic, latency, bandwidth, and computation resources. These computation resources can be further divided into strong and weak worker nodes, where they differ in their computational power in form of CPU power and RAM capacity. Latency values and available bandwidth are adjusted to the specific scenarios, which allows us to mimic real-life network constraints to an adequate level. A more detailed description of the different node types, infrastructure setup/constraints, and network constraints are available in Section 4.2.

In the context of this work, a low constraint environment (both network and resource) corresponds to an IIoT scenario, and the high constraint environment corresponds to an urban sensing scenario. Furthermore, a high resource constraint environment is characterized by edge clusters having more than 50% weak worker nodes. Otherwise, the environment is classified as low constraint. In the context of this thesis, such an environment, together with an orchestration architecture, outlines the setup of the experiments.

Furthermore, each cluster for an orchestration has a different allocation of worker node types. These decisions are based on resource heterogeneity, a recognizable aspect of edge computing. However, since the experiment is conducted on virtual machines, we lose the heterogeneity of CPU resources. Therefore, CPUs in the high constraint setting begin with 50% stress. This artificially created performance degradation allows us to get closer to resource heterogeneity from real-world devices. Network categorization is configured by network latency and available bandwidth in the experimental setup. A highly constrained network has a higher latency than a low one and has a limited bandwidth, while the bandwidth of the low one is not meaningfully limited. Table 5.1 provides a brief overview of the configuration of both constraint levels.

Table 5.1: Configurations of each constrain level

Constrain level	CPU	Initial CPU Stress	RAM	Latency	Bandwidth
high	weak	50%	weak	high	limited
low	strong	0%	strong	low	unlimited

5.1.2 Orchestration Performance Metrics

In order to judge the quality of an orchestration architecture, it is essential to gather information about the current state of the nodes. Therefore, it is essential to find a suitable set of QoS parameters for an empirical experiment [8]. Based on the works [8] and [3], the following part introduces the essential metrics categorized in measurement levels for evaluating the QoS of an orchestration architecture. All the definitions of the used metrics can be found in Table 5.2. We introduce the following levels: system, network, application, and orchestration.

System level

This level represents the physical resources used percentage-wise by the machines. While we analyze the average CPU and RAM usage summarized per cluster and node type, since many nodes are active in the cluster, we do not analyze the CPU or RAM utilization for each node separately. Furthermore, we gathered percentages of CPU and RAM usage for each pod. So, in addition to cluster and node types, we also look at the following pod types per cluster: load-balancer, function replica, scheduler, and autoscaler. For these

Table 5.2: Selected performance metrics

Measurement level	Name (Metric)	Short Description	Abbreviations	Unit	Range
System level	RAM usage	Ram usage relative to the node's capacity.	*ram*	%	0-100
	CPU usage	CPU usage relative to the node's capacity.	*cpu*	%	0-100
Network level	Network throughput	The sum of reads and writes per second in the network.	*net*	MB/s	0-X
	Network latency	Latency values of user requests.	lat	ms	0-X
Application level	Round trip time	Round trip time of user requests.	*rtt*	s	0-X
	Application throughput	Average number of requests processed.	*at*	#	0-X
Orchestration level	Zone crossings	Number of times a request got processed in a different zone than requested from.	*zc*	#	0-X
	Scheduling duration	The time that a scheduling operation took.	*schedd*	ms	0-X
	Scaling duration	The time that a scaling operation took.	*scaled*	ms	0-X
	Times of scaling	The amount of times scaling occurred.	*tos*	#	0-X

pods, the average CPU and RAM usage are examined. Further information about the pod types can be found in Section 4.2.

Network level

This level represents the metrics correlating with in- and outgoing communication from nodes and pods. In our case, network throughput refers to the amount of data sent and received in a set amount of time. Network latency is the time delay between the start and the received timestamp. However, it is essential to mention that the latency value depends on the payload, but since our experiment uses the same payload size for all requests, this is fine.

Regarding the throughput, we analyze the distribution of reads and writes on infrastructure and average values on pod per cluster level. The pod types we look into are the same as the ones on the system level. Regarding network latency, we decided to look at the distribution of latency values for all request paths.

Application level

Application-level metrics depend on the application running. Therefore, we are interested in our application's round trip time and the application throughput the architectures can produce. Round trip time refers to the duration of a request to be processed by the system and sent back to the user. In our case, application throughput refers to the number of answers received per interval. To better understand the application throughput, we also consider the amount of sends per time interval.

Similar to network latency, the round trip time is also analyzed by the distribution of values for all request paths. For the application throughput, we examine average values only for each orchestration architecture in their environments.

Orchestration level

The metrics on the orchestration level refer to orchestration actions. The zone crossings metric refers to the number of times a request got processed in a zone different from the one the user is in. This metric is crucial regarding operational costs since inter-zone communication is often more expensive than intra-zone communication. With the times of scaling metric, we can analyze the resource efficiency of the orchestration architectures. The metrics scaling duration and scheduling duration give us an insight into how fast an architecture can make orchestration decisions. All these metrics are examined for each orchestration architecture.

5.2 Methodology

This chapter provides an overview of configuring the framework for our experiments. Furthermore, it includes how the framework collects the required metrics and how we process them.

5.2.1 Experimental Setup

This section shares more information about how we set up our experiments. It follows a similar structure as Section 4.2 to trace which configurations belong to which area of the framework.

1. Set up VMs

In the context of our experiments, we differentiate between weak and strong VMs. See Table 5.3 for further information. The chosen units are based on [60]. Since one of the

Table 5.3: VM types

VM type	CPU	RAM
Weak	2x Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz	3072 MB
Strong	8x Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz	16384 MB

categorization aspects of edge cloud systems is the resource constraint, we created two cluster setups for each orchestration architecture. Figure 5.2 depicts the infrastructure for high resource constraint environments with inter-network connections. We follow the categorization we defined in Section 5.1.1, where every edge cluster containing more than 50 % of weak worker nodes is considered highly constrained, and each cluster has a different allocation of worker nodes. Figure 5.3 depicts the infrastructure for low-resource environments and inter-network connections. Additionally, the cluster names only have semantic value. In the framework, the zone names divide the infrastructure into the three clusters illustrated in Figure 5.2 and Figure 5.3.

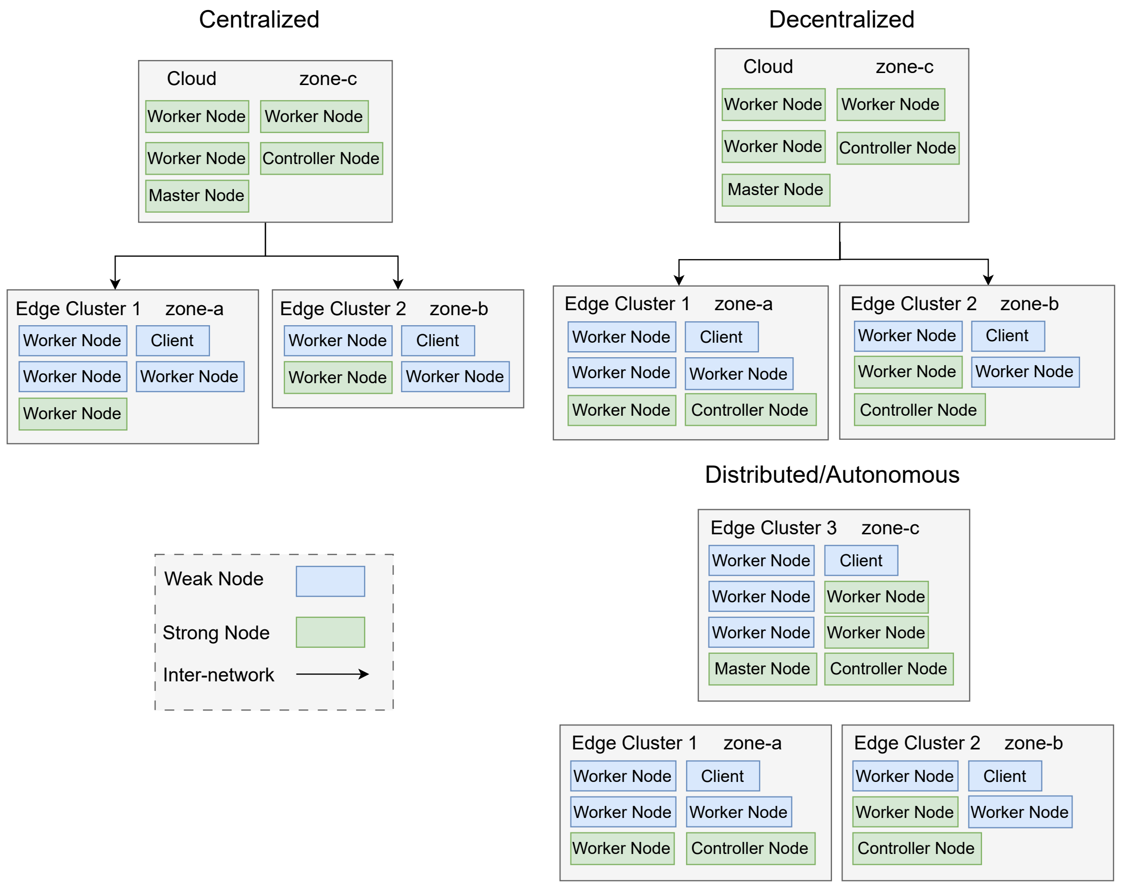


Figure 5.2: High resource constraint infrastructure with inter-network connection for centralized, decentralized and distributed orchestration

Regarding the naming scheme, each node has its corresponding zone and purpose in its name, allowing us, in further steps, to execute tasks specific to zones and node assignments.

2. Install K3s

In this step, the provided Ansible setup gets executed, thereby creating the cluster. Furthermore, we extended the playbook by a post-installation role, which takes care of adding custom Kubernetes labels and authorizing custom controllers to access Kubernetes

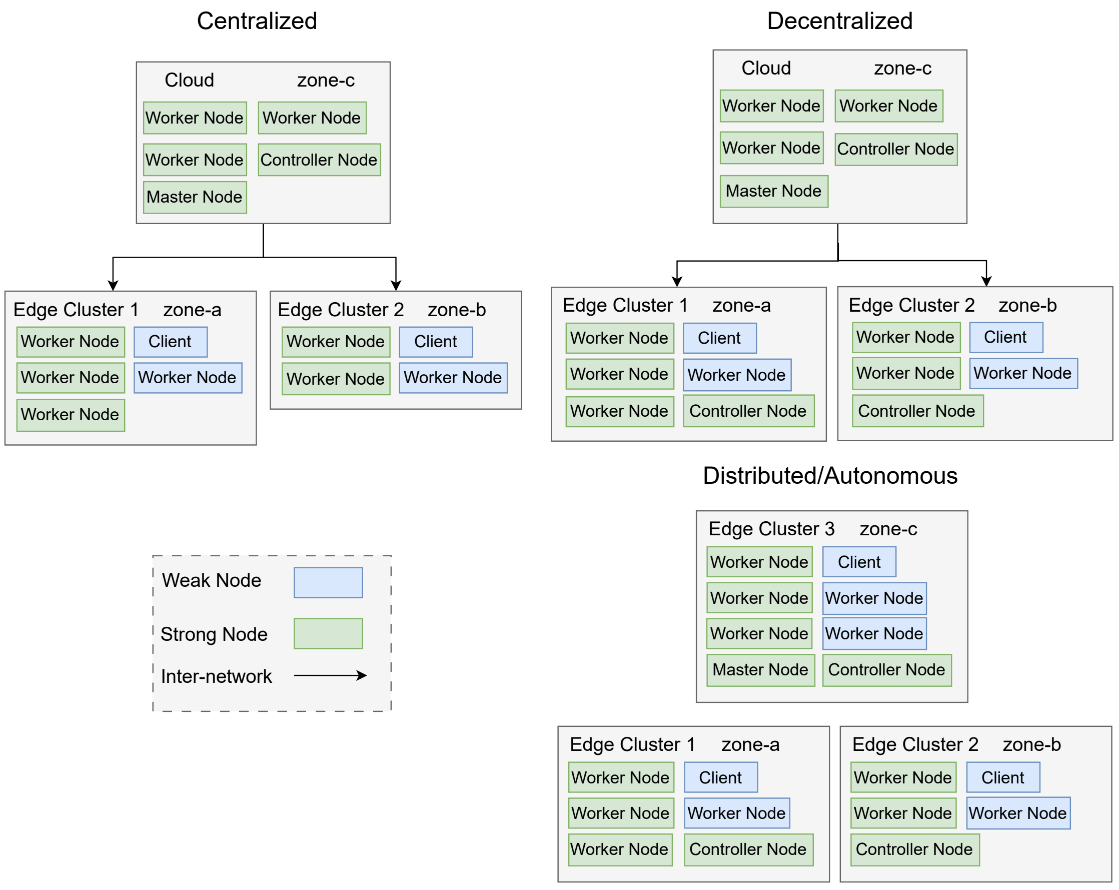


Figure 5.3: Low resource constraint infrastructure with inter-network connection for centralized, decentralized and distributed orchestration

resources using the Kubernetes API. The custom labels define the role of each node in our cluster and are set based on the node name. Regarding node role assignment, we distinguish between four different labels:

- **node-role.kubernetes.io/client=true**
  Nodes with "client" in their name contain this label. They create the workload for the application. The workload consists of simple REST requests with an image as a payload. Each of the requests has an average size of 621 KB.

- **node-role.kubernetes.io/controller=true**
  This label denotes nodes with "controller" in their name. They serve as controllers where orchestration actions such as scheduling and scaling are computed.

- **node-role.kubernetes.io/worker=true**
  Nodes with "worker" in their name have this label and contain pods with the application function running. The function consists of a pi digits calculator.

- **ether.edgerun.io/zone=zone-x**
  Every node has zone-a, zone-b, or zone-c in its name; based on the node name, each node gets a zone label. For example, a node containing the name *zone-a* has the label *ether.edgerun.io/zone=zone-a*. All nodes in one zone are treated as one cluster.

It is essential to mention that since we use Kubernetes with one master node and its Kubernetes API, we cannot build a fully distributed infrastructure because all API communication goes through the master node. For this to be possible, multiple detached master nodes would be needed. Nevertheless, our evaluation is unaffected by this.

3. Deploy Components

We create our storage VM following the instructions from the Galileo framework. Afterward, the control components are deployed when the clusters are up and running. For the centralized architecture, we deploy a global autoscaler, global load-balancer, and global scheduler in the Cloud cluster. For the distributed architecture, each cluster gets a local autoscaler, local load-balancer, and local scheduler. For the decentralized architecture, we deploy one global scheduler in the Cloud cluster, and an autoscaler, a load-balancer, and a local scheduler are deployed in all three clusters. The autoscaler and load-balancer in the decentralized architecture combines the global and local setup. The autoscalers get the information about all clusters but can only scale in their cluster. The load-balancers process requests from clients from their cluster or requests that are derived from load-balancers from another cluster.

Since we use a relatively low number of nodes and simplistic algorithms, the scheduling and autoscaling decision delay is rather non-impactful. We conducted experiments with multiple parameter sets to get closer to real-world applications and depict the impact of cascading delays during orchestration. In order to create meaningful values for the parameter sets, the scenarios and constraint-level setups in which the parameters are used were considered. Table 5.4 describes the three different parameter sets. Furthermore, to better compare the results of the experiments, the target duration and the max scale amount for all experiments are set equal. The target duration conforms to a user request's desired round trip time value in seconds. Moreover, regarding the max scale value, while global autoscalers were given the total max scale value, local autoscalers only allowed a third of the maximum value. This way, we stay within the maximum replica amount since we always have three local autoscalers or one global autoscaler.

Table 5.4: Parameter sets

When used	Abbreviation	Delay	Reconcile interval	Max scale	Target duration
High constraint environment	*hc*	0.5	6	90	0.3
Low constraint environment	*lc*	0.1	2	90	0.3
High and Low constraint environment	*hc_lc*	0.25	4	90	0.3

Table 5.5: Network settings

Network Constrain Level	Latency Edge-Cloud	Latency Edge-Edge	Latency Same Zone	Bandwidth
high	90ms	45ms	15ms	100 Mbps
low	60ms	30ms	10ms	1Gbps

One of the key differences between the infrastructures is the latency between nodes (see Table 5.1). The latency values are based on [58]. In a low network constraint environment, the latency between edge and cloud is 60ms, between the edge clusters 30ms, and nodes inside the same cluster 10ms. Regarding the high network constraint environment, we increase both values by 50 %. Consequently, the values are 90ms, 45ms, and 15ms. Concerning the available bandwidth, in the high network constraint environment, it amounts to 100 Mbps, which corresponds to a Raspberry Pi 3 [40], since Raspberry Pis can often be found in the edge continuum [60, 28]. In a low network constraint environment, we do not limit the bandwidth. The network settings are represented in Table 5.5.

4. Conduct Experiment

We conduct six experiments, analyzing each orchestration architecture in an IIoT and urban sensing scenario, following the categorization defined in Section 5.1.1. Each galileo worker generates workload in a sine wave pattern, as depicted by Figure 5.4. Similar to [58], we use a simple application to mimic an inference task and enable more flexibility for the experiments. Therefore, our experiments run an application that calculates 4000 digits of pi. The usage requirements of the function ($f_{CPU}^{req}, f_{RAM}^{req}$) are 500 milliCPU and 250 megabytes.

All desired metrics (see Section 5.1.2) are collected during the experiments. After the experiment, the sought-after data gets loaded into a Jupyter Notebook, where further data processing occurs.

5.2.2 Metric Data Processing

On the one hand, the data from the telemd service and on the other hand, HTTP traces are used for processing the collected data. Telemd retrieves resource usage relevant data, such as *cpu, ram*, and *net* metrics. A dedicated service from the Galileo environment exists for the metric *tos*. The remaining metrics from Table 5.2 are collected by information from the HTTP traces. Some of the metrics are also analyzed at the zone level. Since we labeled each node with a zone label, it was easy to match each node to its zone.

System level measurements

Telemd allows us to analyze CPU and RAM usage on both node and pod levels for system-level measurements. Since telemd uses Redis, each metric is accessible through distinguished topics. In order to gather the average CPU usage per cluster, we use the

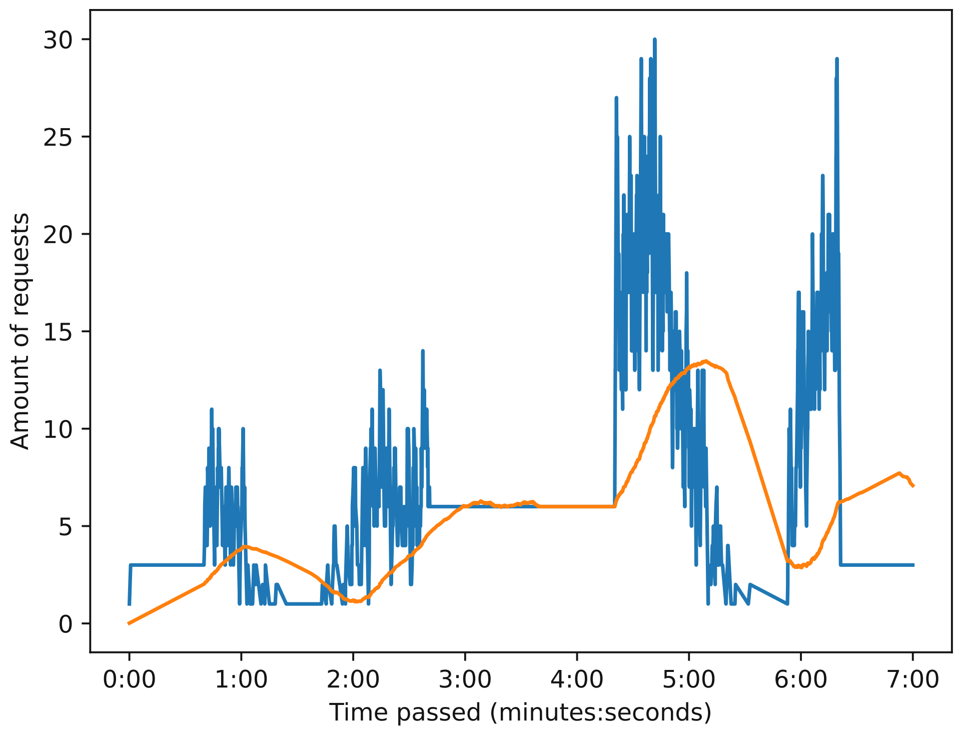


Figure 5.4: User request profile

cpu topic from telemd, which gives us the CPU utilization of the last 5 seconds in %. Regarding RAM usage, telemd only provides us with currently used RAM in kilobytes in its *ram* topic. Therefore, further processing is needed to convert these values to the relative usage in %. Similar actions are needed for converting the values returned by the *kubernetes_cgrp_memory*, which returns the total RAM usage of kubernetes pods in bytes. Unlike topic *cpu*, topic *kubernetes_cgrp_cpu* returns the CPU usage time of individual Kubernetes pods. Further preprocessing is needed to turn these CPU usage time values into an average usage.

Network level measurements

For metric *net* on the orchestration level, telemd provides the tx and rx bit rate of each node for each subsystem through the *rx* and *tx* topics. In this case, preprocessing, in the form of summing these values together, was needed to reach the desired metric. Topic *kubernetes_cgrp_net*, which we use for pod-level analysis, already provides the sum of the tx and rx bit rates. Metric *lat* is calculated using the information provided by the HTTP request traces. No preprocessing is needed since the traces directly provide the latency value for each request and their origin and destination cluster.

Application level measurements

For analyzing the *rtt*, no additional preprocessing is needed since all information is provided directly in the traces, similar to *lat*. Regarding *at*, since traces already provide us with timestamps for finished requests, it is simple to prepare the *at* metric. Additionally, the timestamps for sending requests are also examined for better comparison.

Orchestration level

Regarding *zc*, here as well, all information is directly extractable from the traces, so no real preprocessing is needed. Since telemd does not support *schedd* nor *scaled* natively, we used a telemd feature, which allows us to create custom topics. Therefore, for *schedd*, we measured the duration of the scheduling process and published it into a custom-created topic, which can be accessed by telemd. Something similar was done for *scaled*. Finally, for *tos* metric, we can utilize a service from the Galileo benchmark tool, which allows us to extract the number of running replicas during an experiment.

5.3 Results

The following section contains a comprehensive and detailed analysis that serves as the base for the recommendation in Chapter 6. It displays and describes the collected metrics throughout our experiments based on the used parameters. For the parameter sets, *hc* and *lc*, the different orchestration architectures within the same constrained infrastructure were used. Results of experiments conducted with parameter set *hc_lc* had the same orchestration architecture in different constrained environments. Furthermore, for all following plots, <architecture_hc> and <architecture_lc> stand for what architecture and level of infrastructure constraint was used, where hc stands for high constraint and lc for low constraint. For the bar graphs, average values are illustrated, where the black bars depict the 95% confidence interval of the true mean. Shadows in the progress graphs also stand for the 95% confidence interval of the true mean. The standard deviations of each bar graph's highest and lowest mean values are mentioned in the subsections below.

5.3.1 System level *hc* parameter set

Figure 5.5 depicts the average CPU and RAM usage for each cluster and architecture in a high constraint environment. In centralized_hc, the highest average CPU usage was measured in Edge-Cluster 1, with a CPU usage of 50.8% and a standard deviation of 20.9%. The lowest usage was found in the Cloud, with a value of 23.2% and a standard deviation of 11.6%. Memory usage-wise, Edge-Cluster 1 also had the highest RAM usage of 16.4% with a standard deviation of 5.3%. The lowest RAM usage goes to the Cloud with a value of 7.8% and a standard deviation of 1%.

In decentralized_hc, the highest average amount of CPU usage is found in Edge-Cluster 1, with a value of 47.4% and a standard deviation of 19.3%. Similarly to centralized_hc, also in decentralized_hc, the lowest average CPU usage was measured in the Cloud with a value of 10.7% and a standard deviation of 12.9%. Regarding RAM usage, the highest measured usage was found in Edge-Cluster 1 with a value of 14.4% and a standard deviation of 5.5%, and the lowest in the Cloud with a usage of 5.4% and a standard deviation of 1.8% .

Similarly to decentralized_hc and centralized_hc, for the distributed_hc experiments, the highest measured amount of average CPU usage and average RAM usage was also found in Edge-Cluster 1. The average CPU usage was 54.6% with a standard deviation of 22.9%, and the average RAM usage was 15.5% with a standard deviation of 6.5%. In Edge-Cluster 3, the lowest CPU usage was measured with a value of 44.4% and a standard deviation of 20.7%, and the lowest RAM usage with a value of 12.3% and a standard deviation of 4.8%.

Figure 5.6 presents the average CPU and RAM usage for each node type and architecture in high constraint environments. In all three architectures, the average CPU usage of weak nodes was close to each other. The highest amount was measured in distribtued_hc with 67.7% and a standard deviation of 11.1%. Decentralized_hc had the lowest value with 60.7% and a standard deviation of 7.9%. We have a similar situation for the strong nodes, where the distributed_hc strong nodes had the highest CPU usage with 29.1% and a standard deviation of 13.7%. The decentralized_hc strong nodes had the lowest usage, with 18.8% average CPU usage and a standard deviation of 15%.

Regarding average RAM usage, centralized_hc and distributed_hc had the highest RAM usage on weak nodes. Both had an average usage of 19.3%, where centralized had a standard deviation of 2% and distributed_hc a standard deviation of 2.2%. Therefore, decentralized_hc had the lowest average RAM usage for weak nodes, with 18.7% and a standard deviation of 1.3%. On strong nodes, both centralized_hc and distributed_hc had the highest average RAM usage of 7.7%. Centralized_hc had a standard deviation of 1% and distributed_hc a standard deviation of 0.9%. Similarly to the weak nodes, also with the strong nodes, the decentralized_hc infrastructure has the lowest average RAM usage, with a value of 6.6% and a standard deviation of 1.8%

Figure 5.7 illustrates the average CPU usage of an autoscaler for each cluster. In centralized_hc, the autoscaler had a consumption of 1% and a standard deviation of 0.21%. The CPU usage of autoscalers in decentralized_hc were all very close. The autoscaler with the highest consumption was Edge-Cluster 2, with a value of 0.94% and a standard deviation of 0.18%. The lowest was in the cloud, with 0.91% average CPU usage and a standard deviation of 0.18%. Also, for distributed_hc, the CPU usage was very similar across the clusters. The highest was measured in Edge-Cluster 2 with a value of 1.64% and a standard deviation of 0.52%, and the lowest in Edge-Cluster 3 with a value of 1.58% and a standard deviation of 0.45%.

As Figure 5.8 depicts, in centralized_hc, the average RAM usage of an autoscaler was 0.68% with a standard deviation of 0.04%. In decentralized_hc, all autoscalers of each cluster had a very similar RAM consumption. The highest was at 0.7% with a standard deviation of 0.04%, and the lowest was at 0.68% with a standard deviation of 0.03%.

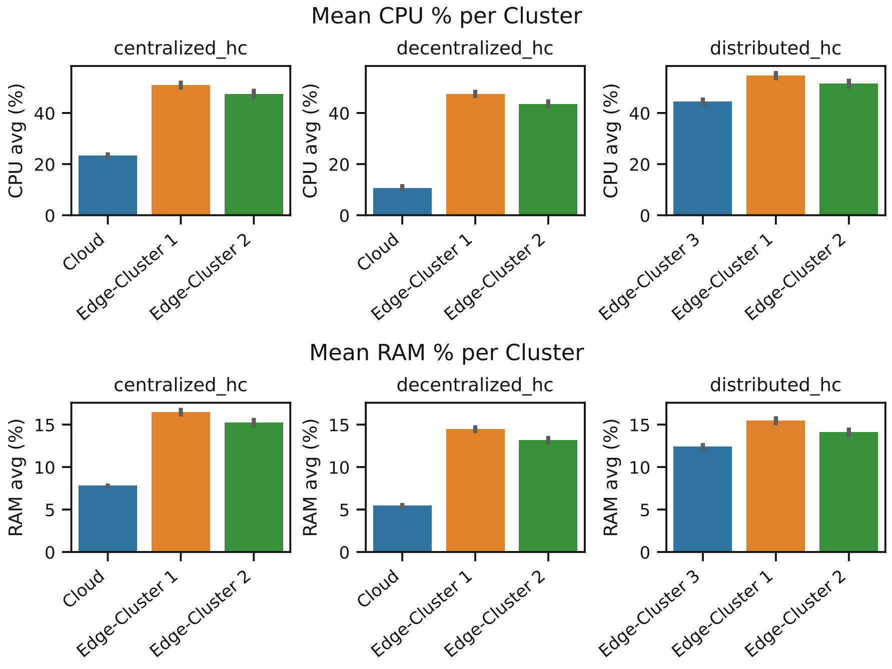


Figure 5.5: Average CPU and RAM usage per Cluster, high constraint infrastructure, all architectures

The autoscalers in distributed_hc also all had a nearly identical average RAM usage of 0.7% and a standard deviation of 0.05%.

In Figure 5.9, we can see the average CPU usage of a function replica throughout the experiments. 0.56% with a standard deviation of 0.55% is the highest average CPU usage in centralized_hc. The lowest one is 0.38% with a standard deviation of 0.35%. In decentralized_hc, only Edge-Cluster 1 and 2 had function replicas running. In Edge-Cluster 1, we measured an average CPU usage of 1.4% with a standard deviation of 1.6%; in Edge-Cluster 2, a usage of 1.5% and a standard deviation of 1.8%. The Edge-Cluster 1 in distribtued_hc measured the highest average CPU usage in the distributed architecture, with a value of 2.7% and a standard deviation of 4.5%. Edge-Cluster 3 had the lowest measured amount with 0.73% and a standard deviation of 0.65%.

The following figure, Figure 5.10, describes the average RAM usage of a function replica. Looking at the centralized_hc architecture, we can see that the function replicas in Edge-Cluster 1 had, in total, the highest average RAM usage of 0.51% and a standard deviation of 0.26%. The lowest usage was found in the Cloud, with 0.2% and a standard deviation of 0.02%. Similar to Figure 5.9, in this case, the decentralized_hc experiments had only Edge-Cluster 1 and 2 function replicas running. A replica in Edge-Cluster 1 consumed an average of 0.62% of RAM with a standard deviation of 0.07% and in Edge-Cluster 2 0.53% with a standard deviation of 0.09%. In distributed_hc, the highest average RAM usage was the Edge-Cluster 1 with a usage of 0.7% and a standard deviation

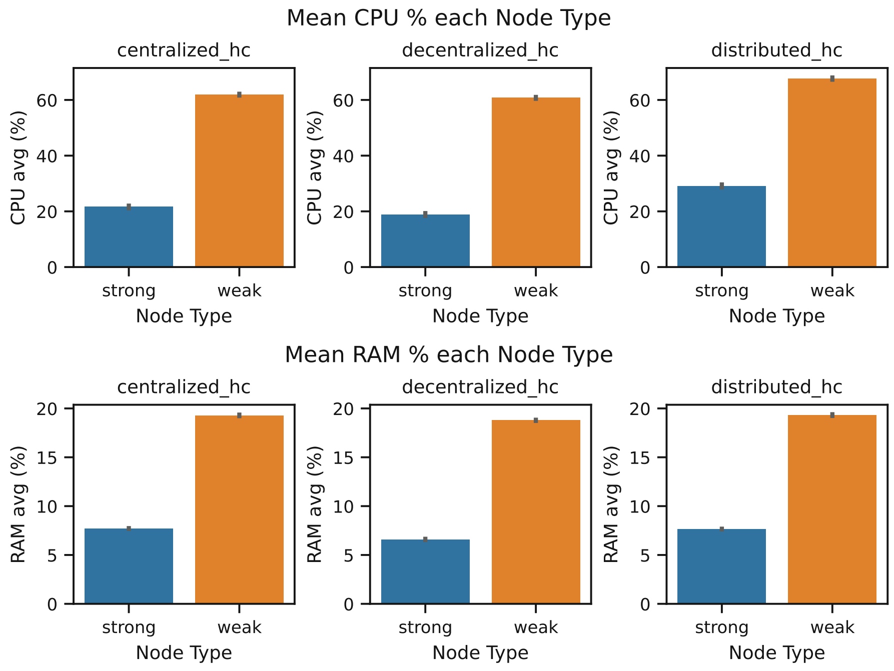


Figure 5.6: Average CPU and RAM usage per Node Type, high constraint infrastructure, all architectures

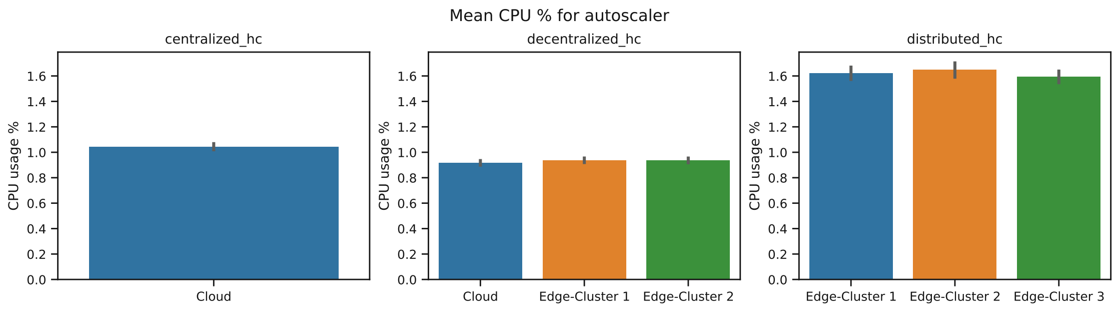


Figure 5.7: Average CPU usage from autoscaler, high constraint infrastructure, all architectures

of 0.09%. The lowest usage was in Edge-Cluster 3, with a value of 0.19% and a standard deviation of 0.02%.

The next component whose CPU and RAM utilization we look at is the load-balancer. Figure 5.11 illustrates the average CPU utilization. The load-balancer in centralized_hc had a usage of 1.8% with a standard deviation of 1%. For decentralized_hc, we measured in Edge-Cluster 1 an average CPU usage of 1.14% with a standard deviation of 1% as the highest value and in the Cloud a 0.01% amount of usage with a standard deviation of 0.002% as the lowest value. The usage in distributed_hc is similarly distributed, where the highest value is at 0.93% with a standard deviation of 0.86% in Edge-Cluster 2, and

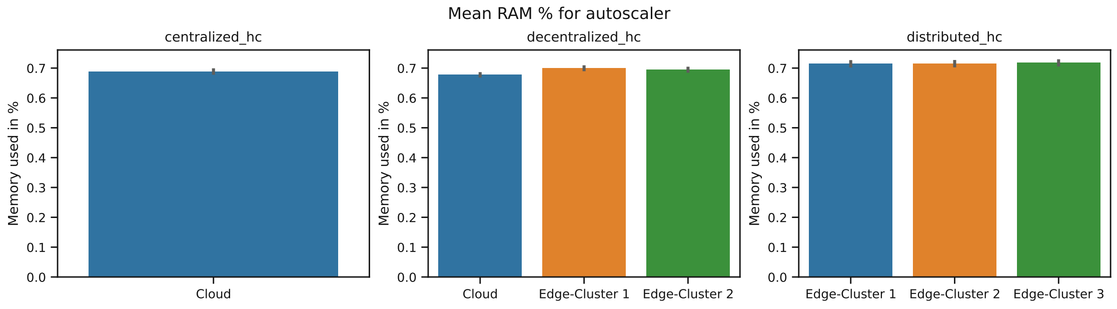


Figure 5.8: Average RAM usage from autoscaler, high constraint infrastructure, all architectures

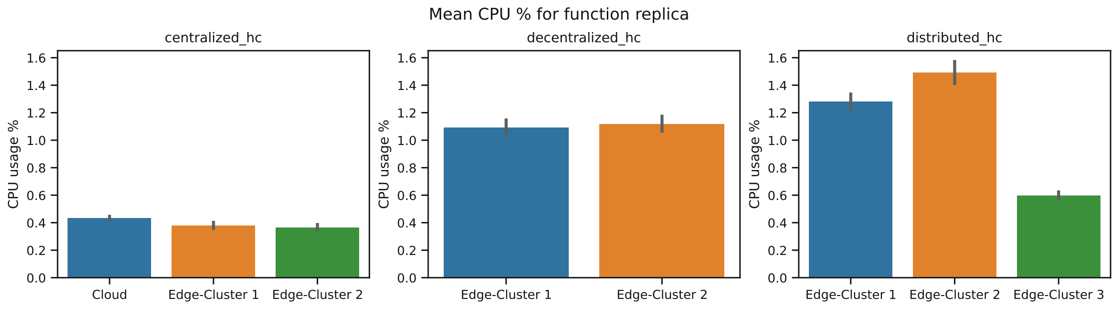


Figure 5.9: Average CPU usage from function replicas, high constraint infrastructure, all architectures

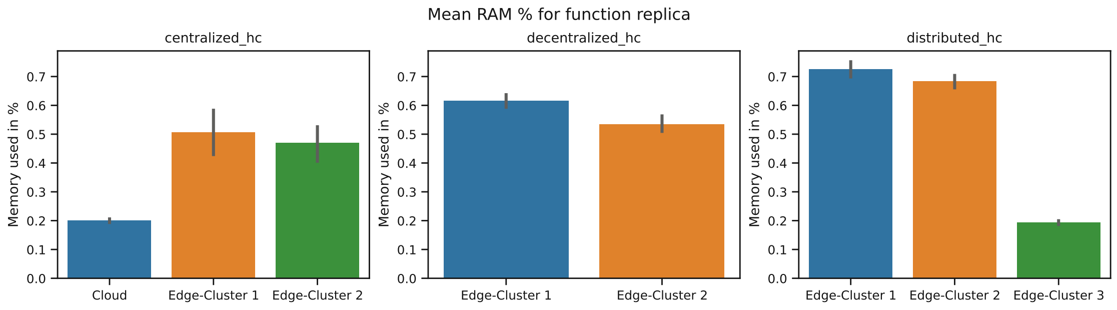


Figure 5.10: Average RAM usage from function replicas, high constraint infrastructure, all architectures

the lowest value of 0.87% with a standard deviation of 0.81% in Edge-Cluster 3.

Figure 5.12 depicts the average RAM usage of a load-balancer. The load-balancer in centralized_hc had an average RAM usage of 0.18% with a standard deviation of 0.11%. Average usage of 0.15%, with a standard deviation of 0.12%, was the highest measured RAM in decentralized_hc in Edge-Cluster 2. The lowest one was in the Cloud, with a mean value of 0.05% and a standard deviation of 0.003%. Similarly to the average CPU usage, the mean RAM usage was evenly distributed in distributed_hc. All had an

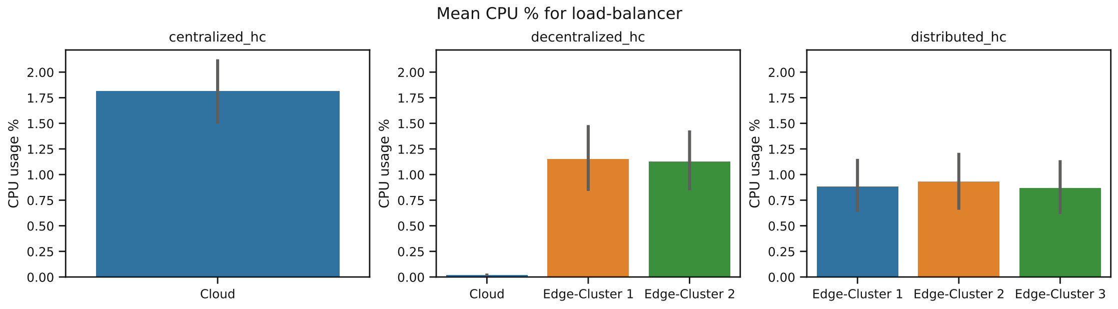


Figure 5.11: Average CPU usage from load-balancer, high constraint infrastructure, all architectures

average usage of 0.09% and a standard deviation of 0.03%.

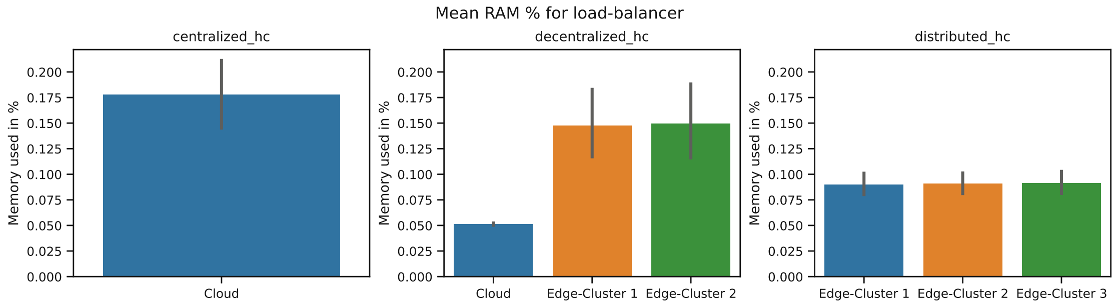


Figure 5.12: Average RAM usage from load-balancer, high constraint infrastructure, all architectures

The last control component we measured was the schedulers. Figure 5.13 illustrates the average CPU usage of a scheduler during the experiments for each high constraint infrastructure for all three architectures. As we can see from the figure, the scheduler of centralized_hc had an average CPU usage of 4.1% with a standard deviation of 4.1%. The average CPU utilization in the decentralized_hc strategy was all pretty even. The highest amount was in the Cloud, with a value of 1.11% and a standard deviation of 0.5%. The lowest one was at Edge-Cluster 2, with an amount of 1.08% and a standard deviation of 0.5%. In distributed_hc, we can see a more significant difference between the clusters. In Edge-Cluster 2, we measured the highest value of 7.6% with a standard deviation of 2.6%, and in Edge-Cluster 3, we measured the lowest value in decentralized_hc, with a value of 3.2% and a standard deviation of 2.2%.

As we can see from Figure 5.14, the average RAM usage was for all infrastructures nearly identical. The highest value was measured in Edge-Cluster 2 from the distributed_hc scenario, with 0.71% and a standard deviation of 0.05%. The lowest value was measured in the Cloud in decentralized_hc, with 0.68% and a standard deviation of 0.03%.

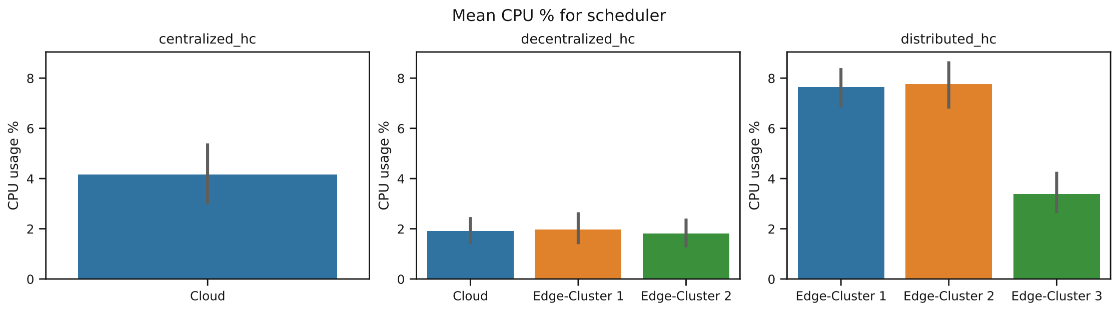


Figure 5.13: Average CPU usage from scheduler, high constraint infrastructure, all architectures

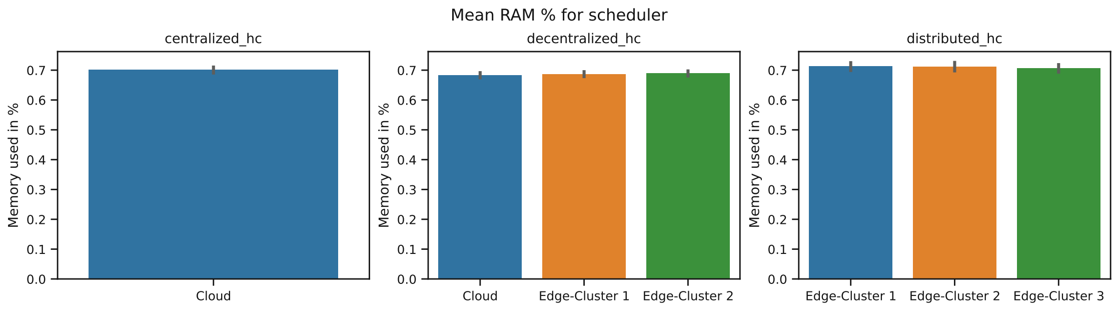


Figure 5.14: Average RAM usage from scheduler, high constraint infrastructure, all architectures

5.3.2 System level *lc* parameter set

In Figure 5.15, we can see each architecture's average CPU and RAM usage for each architecture by their clusters. In centralized_lc, the highest value was in Edge-Cluster 2 with a mean of 29.8% and a standard deviation of 22.9%. The lowest was in the Cloud, with a value of 21.3% and a standard deviation of 12.4%. We can see a similar structure in decentralized_lc. Here, the highest mean is also in Edge-Cluster 2, with a mean value of 33.2% and a standard deviation of 23.8%. The lowest value is in the Cloud as well, with 14.6% and a standard deviation of 15.8%. On the contrary to centralized_lc and decentralized_lc, distributed_lc follows a different pattern. While the highest mean value was found in Edge-Cluster 2 with 32.3% and a standard deviation of 24.2%, the lowest was found in Edge-Cluster 1 with an average value of 26.7% and a standard deviation of 24.1%.

RAM usage-wise, the infrastructures follow a nearly identical pattern as in CPU usage. In centralized_lc, the highest mean value, 10.2%, was in Edge-Cluster 2 with a standard deviation of 5.1%, and the lowest one in the Cloud with an average value of 6.7% and a standard deviation of 4.8%. In the decentralized_lc infrastructure, the highest measured average value was in Edge Cluster-2 with 8.9% with a standard deviation of 5.2%. 5.4%, with a standard deviation of 1.9%, was the lowest mean value in decentralized_lc and was measured in the Cloud. The average RAM usage pattern in distributed _lc differs from its CPU usage pattern. For the average RAM usage, Edge-Cluster 3 has the highest mean value of 9.5% with a standard deviation of 5.5%. The lowest one is in Edge-Cluster 1, with a value of 8.2% and a standard deviation of 5.4%.

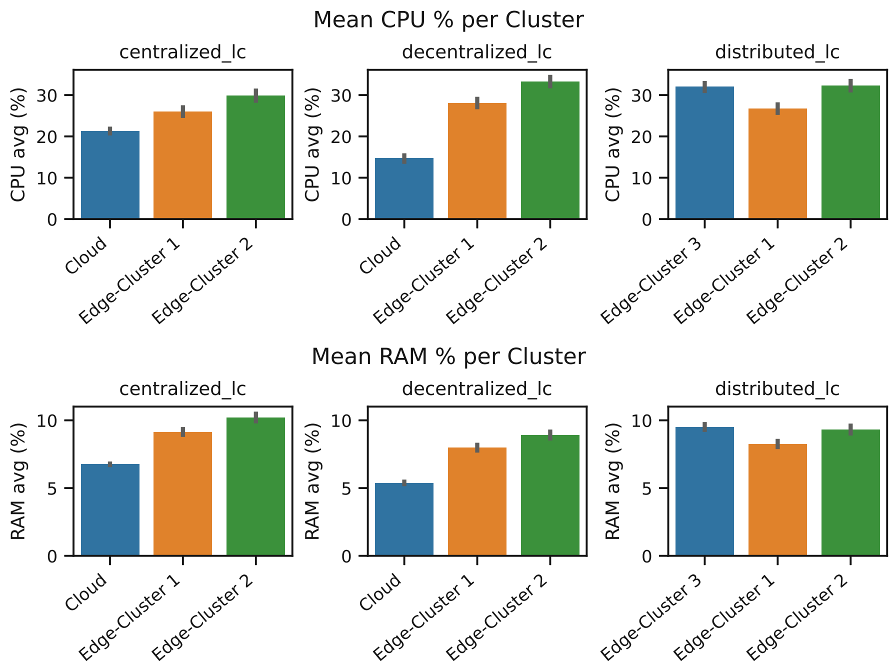


Figure 5.15: Average CPU and RAM usage per Cluster, low constraint infrastructure, all architectures

Figure 5.16 illustrates nodes' mean CPU and RAM usage by their node types. If we look at the CPU usage, we can see that the values for both strong and weak nodes are close. The highest figure for weak nodes, 67.5%, was measured in decentralized_lc with a standard deviation of 9.8%, and the lowest in centralized_lc with a value of 60.5% and a standard deviation of 7.5%. For strong nodes, decentralized_lc also has the highest value, with 17.9% and a standard deviation of 15%. The lowest value was in centralized_lc with 17.5% and a standard deviation of 10.3%.

Regarding average RAM usage in weak nodes, the highest mean value was measured in distributed_lc with 18% and a standard deviation of 0.9%. The lowest average value was in centralized_lc with 17.3% and a standard deviation of 0.3%. For the strong nodes, the highest mean figure was measured in centralized_lc with 6.6% and a standard deviation of 1.1%, and the lowest in decentralized_lc with a mean value of 5.6% and a standard deviation of 1.7%.

Looking at Figure 5.17, we can see the average CPU usage of the autoscaler across all three infrastructures for each architecture. In centralized_lc, the mean usage was 1.2% with a standard deviation of 0.33%. In decentralized_lc, we found the highest mean

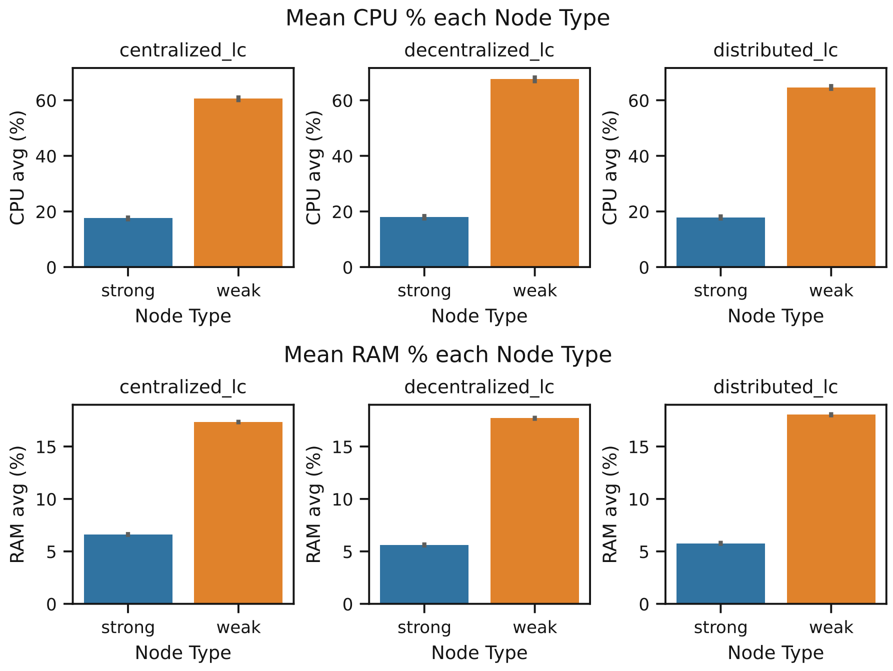


Figure 5.16: Average CPU and RAM usage per Node Type, low constraint infrastructure, all architectures

usage with 1.22% and a standard deviation of 0.37% in Edge-Cluster 1. The Cloud had the lowest mean usage with 1.1% and a standard deviation of 0.3%. In distributed_lc, the highest value was in Edge-Cluster 1 at 1.24% with a standard deviation of 0.63%, and the lowest was in Edge-Cluster 3 at 1.19% with a standard deviation of 0.64%.

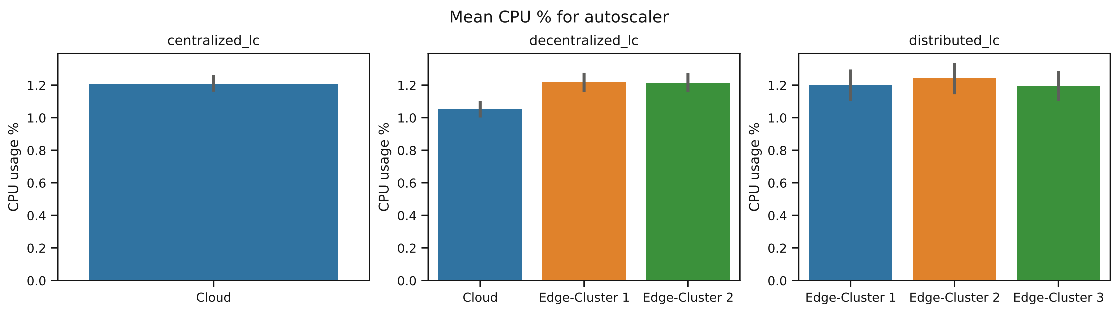


Figure 5.17: Average CPU usage from autoscalers, low constraint infrastructure, all architectures

If we look at Figure 5.18, we can see that an autoscaler in each infrastructure and cluster had a mean usage of around 0.7%. Additionally, the standard deviation of each value was around 0.04%.

Figure 5.19 depicts a function replica's average CPU usage per infrastructure, archi-

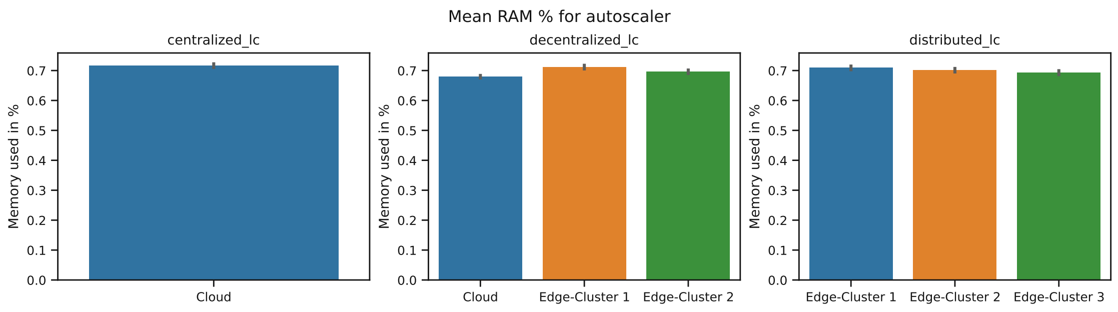


Figure 5.18: Average RAM usage from autoscalers, low constraint infrastructure, all architectures

tecture, and cluster. We can see immediately that they all have a different outcome. In centralized_hc, the highest mean value was in the Cloud with a figure of 0.48% and a standard deviation of 0.52%, and the lowest was found in Edge-Cluster 2. In decentralized_lc, only two clusters, Edge-Cluster 1 and Edge-Cluster 2, had replicas running. The mean CPU usage in Edge-Cluster 1 was 4.5%, with a standard deviation of 3.5%. Edge-Cluster 2 had a mean usage of 3.8% and a standard deviation of 2.4%. The highest measured average value in distributed_lc was Edge-Cluster 2 with 4.7% and a standard deviation of 7.2%. 3.5% in Edge-Cluster 3 was the lowest measured average value and had a standard deviation of 2.7%.

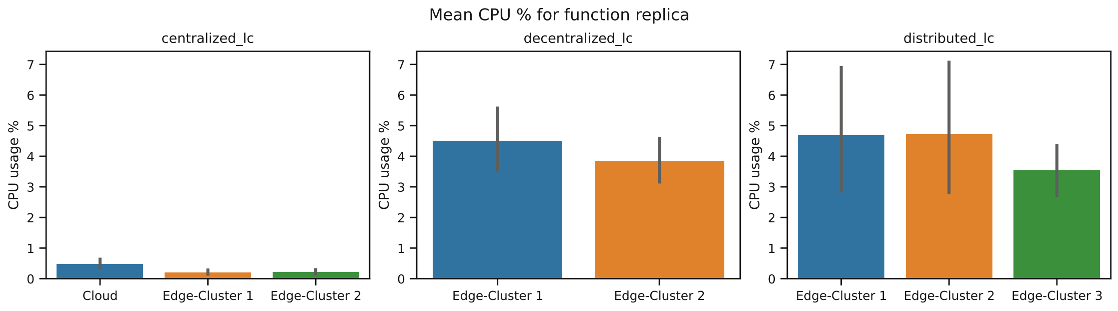


Figure 5.19: Average CPU usage from function replicas, low constraint infrastructure, all architectures

The following figure, Figure 5.20, illustrates the average RAM usage in the same way as Figure 5.19. In centralized_lc, the usage was nearly identical over the three infrastructures with a value of around 0.2% and standard deviation at 0.2%. Similarly, Function replicas from Edge-Cluster 1 and Edge-Cluster 2 from decentralized_lc also had nearly equal mean RAM usage at around 0.5%. The standard deviation in Edge-Cluster 1 was at 0.32%, and in Edge-Cluster 2 was at 0.22%. On the contrary to decentralized_lc and centralized_lc, in distributed_lc, there are more significant differences between the clusters. The highest measured mean RAM usage was in Edge-Cluster 1, with a value of 0.6% and a standard deviation of 0.13%. The lowest measured value, 0.21%, was measured in Edge-Cluster 3 and had a standard deviation of 0.03%.

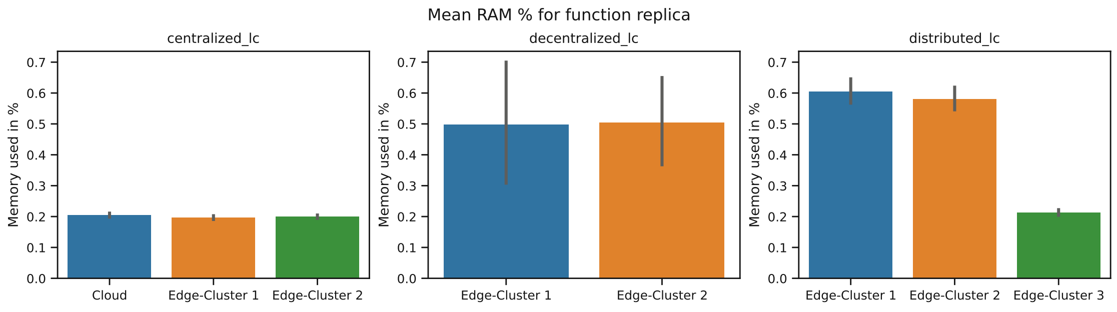


Figure 5.20: Average RAM usage from function replicas, low constraint infrastructure, all architectures

Figure 5.21 describes the average CPU usage of the load-balancer for each cluster in low constraint infrastructures with all architectures. In centralized_lc, the mean value was 0.87% with a standard deviation of 0.91% in the Cloud. In decentralized_lc, the highest measured mean figure was in Edge-Cluster 1 with a value of 1.1% and a standard deviation of 1.6%. The lowest measured average value was in the Cloud with 0.01% and a standard deviation of 0%. If we look at distributed_lc, we can see that each load-balancer from each cluster has nearly the same average CPU usage of 0.4% with a standard deviation of around 0.3%.

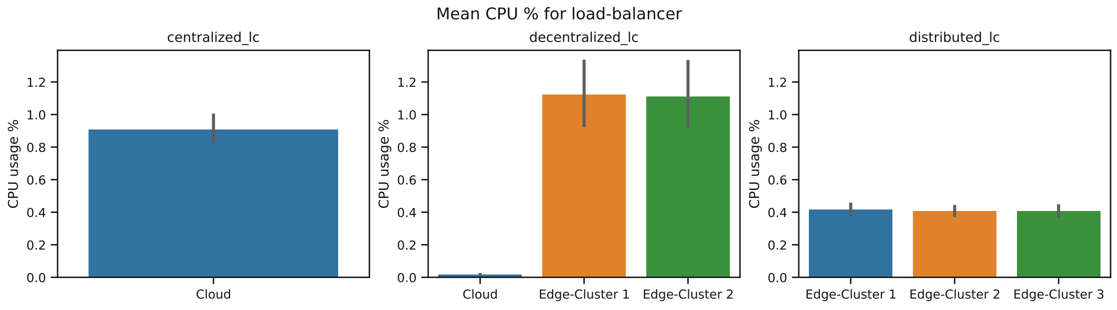


Figure 5.21: Average CPU usage from load-balancers, low constraint infrastructure, all architectures

Figure 5.22 depicts the average RAM usage of the load-balancers. In the Cloud of the centralized_lc experiment, the load-balancer had a mean RAM usage of 0.13% and a standard deviation of 0.05%. In decentralized_lc, in Edge-Cluster 1, the highest figure of 0.23% with a standard deviation of 0.21% was measured. The load-balancer in the Cloud had the lowest figure of 0.05% with a standard deviation of 0%. Similar to the mean CPU usage, the mean RAM usage is also closely even among the clusters in distributed_lc. The mean value hovers around 0.08% with a standard deviation of around 0.02%.

The subsequent two figures, Figure 5.23 and Figure 5.24, illustrate the schedulers' average CPU and RAM usage per cluster, per architecture for low constraint infrastructures.

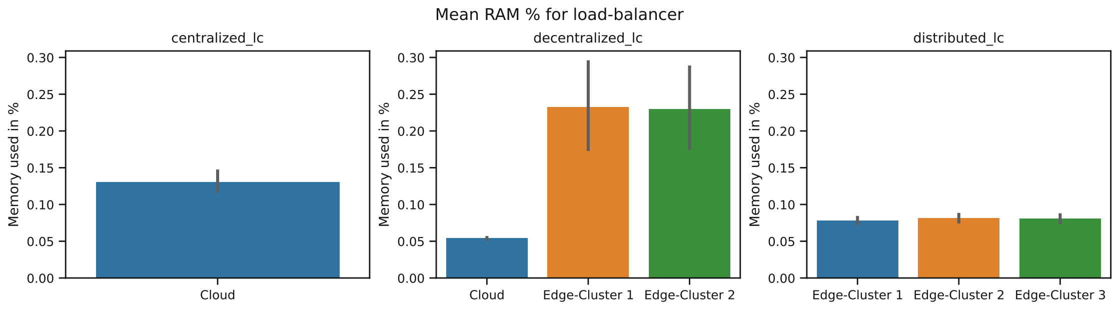


Figure 5.22: Average RAM usage from load-balancers, low constraint infrastructure, all architectures

The mean CPU usage of the scheduler in the Cloud in centralized_lc was at 5.7% with a standard deviation of 4.6%. In decentralized_lc, the highest measured mean usage was in Edge-Cluster 1 with a value of 5.3% and a standard deviation of 4%, and the lowest value with 5% and a standard deviation of 3.8% was measured in Edge-Cluster 2. If we look at the data collected during the distributed_lc experiments, we found that the highest average CPU usage was in Edge-Cluster 1 with a value of 3.2% and a standard deviation of 3.9%. The lowest usage was found in Edge-Cluster 3 with 3% and a standard deviation of 3.8%.

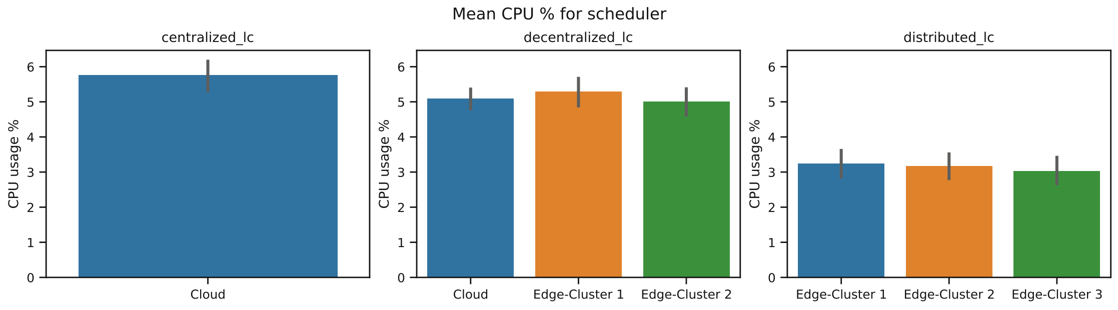


Figure 5.23: Average CPU usage from schedulers, low constraint infrastructure, all architectures

As mentioned above, Figure 5.24 describes the average RAM usage of schedulers. In centralized_hc, the mean usage was 0.7% with a standard deviation of 0.03%. The schedulers in all three clusters in decentrlizaed_hc had a nearly identical mean RAM usage of 0.69% and a standard deviation of 0.03%. Similarly, in distribtued_hc, all schedulers had a mean RAM usage of 0.7% with a standard deviation of 0.04%.

5.3.3 System level *hc_lc* parameter set with centralized architecture

The following figure, Figure 5.25, illustrates the average CPU and RAM usage per cluster for the centralized architecture used in the high and low constraint environment. For the low constraint environment, the highest mean value was in Edge-Cluster 2 with 29.7%

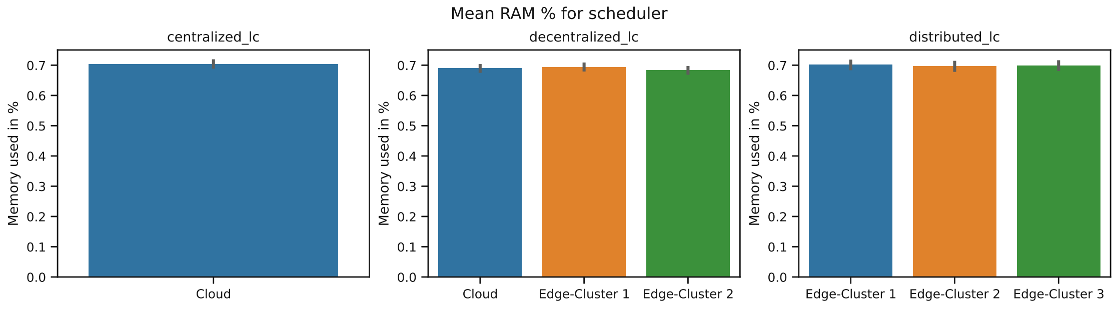


Figure 5.24: Average RAM usage from schedulers, low constraint infrastructure, all architectures

and a standard deviation of 22.3%. The lowest one was in the Cloud with 21.4% and a standard deviation of 12%. In the low constraint environment, the highest one was in Edge-Cluster 1, with a value of 49.7% and a standard deviation of 21.6%. Similarly to the low constraint environment, the high constraint had the Cloud with the lowest average CPU usage at 22.2% and a standard deviation of 11.5%.

The same pattern holds for average RAM usage in both environments. In the low constraint environment, the highest average RAM usage was in Edge-Cluster 2 with 10% and a standard deviation of 5.1%, and the lowest in the Cloud with 6.6% usage and a standard deviation of 1.1%. For the high constraint environment, the highest mean usage was at 16.1% with a standard deviation of 5.2% in Edge-Cluster 1. 7.6% and a standard deviation of 1.1% was the lowest average RAM usage measured in the Cloud.

Figure 5.26 depicts nodes' average CPU and RAM usage for low and high constraint environments, categorized by node type per cluster. In this case, the average RAM and CPU usage of both node types were very close for both environments. The mean CPU usage for weak nodes in the low constraint environment was at 60% with a standard deviation of 7.3%, and in the high constraint environment, it was at 60.6% with a standard deviation of 6.5%. For the strong nodes, the mean CPU usage in the low constraint environment was 18% with a standard deviation of 10%. In the other environment, the usage was at 20.5% with a standard deviation of 11.2%.

We had a similar situation regarding the mean RAM usage. In the low constraint environment, the value for the weak nodes was at 17% with a standard deviation of 0.3%, and in the high constraint environment, at 19% with a standard deviation of 1.6%. Regarding strong nodes, the value in the low constraint environment was at 6.6% with a standard deviation of 1.1%, and in the high constraint environment at 7.6% with a standard deviation of 1.1%.

The following two figures, Figure 5.27 and Figure 5.28 illustrate the average CPU and RAM usage of the autoscaler in low and high constraint environment with the centralized architectures. Both mean CPU and RAM usage were very similar. The usage in the low constraint environment was at 1.1% with a standard deviation of 0.3%, and in the high

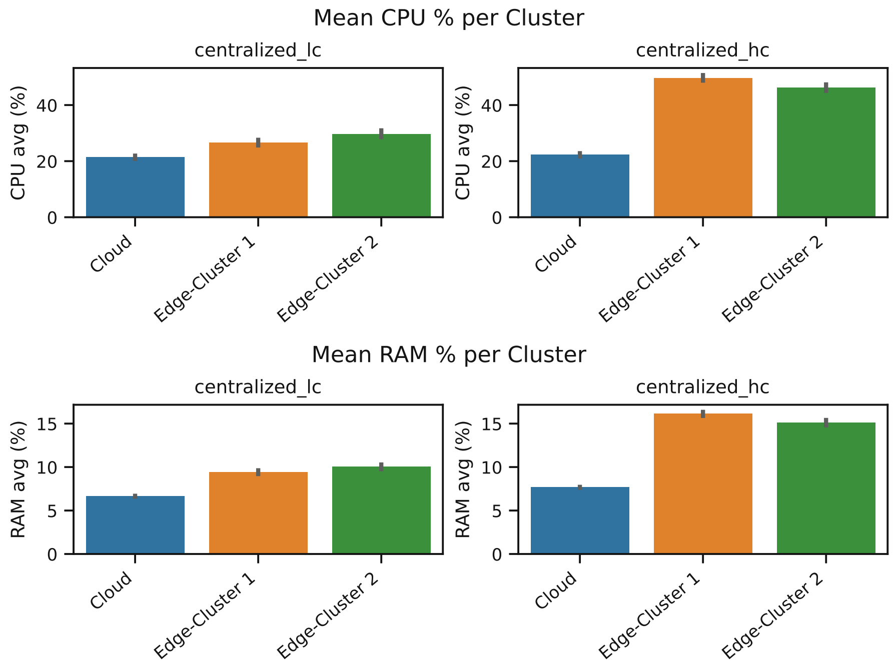


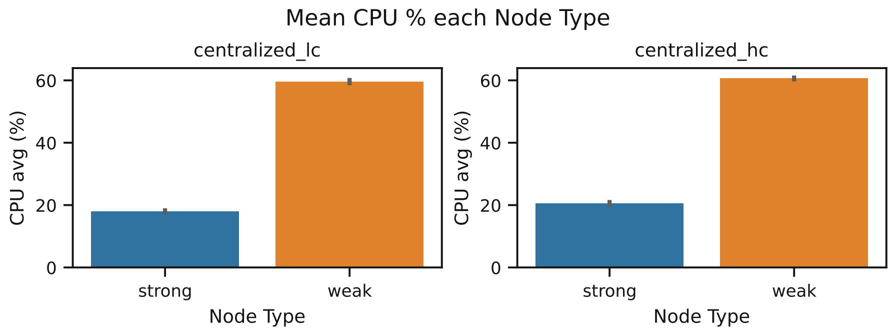
Figure 5.25: Average CPU and RAM usage per Cluster, high and low constraint environment, centralized architecture

constraint, 1% with a standard deviation of 0.23%. Regarding average RAM usage, in both environments, the was at around 0.7% with a standard deviation of 0.05%.

Figure 5.29 describes the average CPU usage of a function replica per cluster for both constraint environments with a centralized architecture. In both environments, the mean CPU usage value was highest in the Cloud and lowest in Edge-Cluster 2. In centralized_lc, the highest value was at 0.55% with a standard deviation of 0.57%. In centralized_hc, it was at 0.37% with a standard deviation of 0.5%. The lowest mean value in centralized_lc was at 0.22% with a standard deviation of 0.22%, and 0.3% with a standard deviation of 0.28% in centralized_hc. If we look at Figure 5.30, we can see that the average RAM usage in centralized_lc was nearly identical, with a value of 0.2% and a standard deviation of 0.02%. In centralized_hc, only Edge-Cluster 1 and Edge-Cluster 2 had a similar mean usage of 0.44% and a standard deviation of 0.18. Cloud only had a mean usage of 0.2% with a standard deviation of 0.02%.

Looking at Figure 5.31 and Figure 5.32, we see that both average CPU and RAM usage was higher in centralized_hc than in centralized_lc. For CPU usage, the higher mean value was 1.3% with a standard deviation of 0.9%, and the lower value was 0.97% with a standard deviation of 0.9%. Regarding mean RAM usage, in centralized_hc, the value was at 0.2% with a standard deviation of 0.1%, and in centralized_lc at 0.12% with a standard deviation of 0.05%.

Figure 5.33 illustrates the average CPU usage of the scheduler in both infrastructures



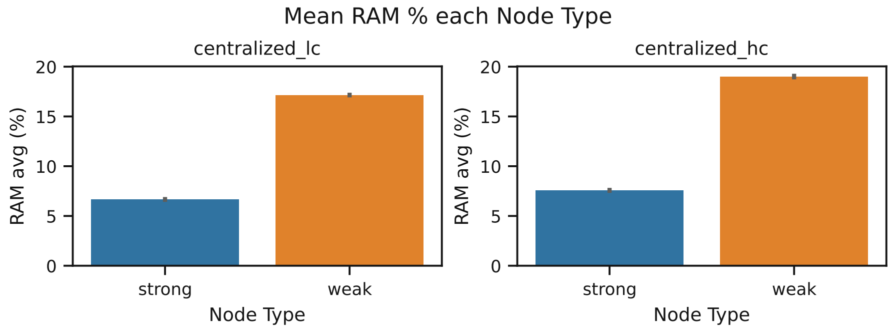


Figure 5.26: Average CPU and RAM usage per Node Type, high and low constraint environment, centralized architecture

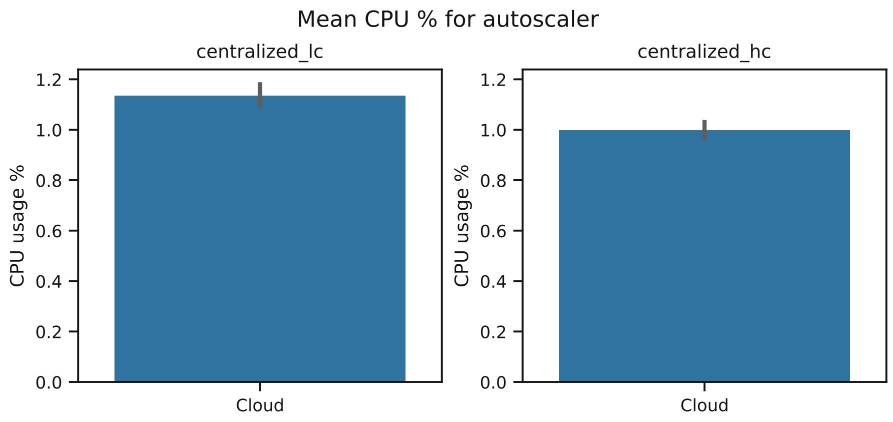


Figure 5.27: Average CPU usage from autoscaler, high and low constraint environment, centralized architecture

using a centralized architecture. We see that the usage in centralized_lc was higher than in centralized_hc. The higher value was at 5.5% with a standard deviation of 4.5%, and the lower value was at 4.2% with a standard deviation of 4%. Figure 5.24 depicts the average RAM in the same way was Figure 5.33. The scheduler in both infrastructures had a nearly identical mean RAM usage of 0.7% with a standard deviation of around 0.04%.

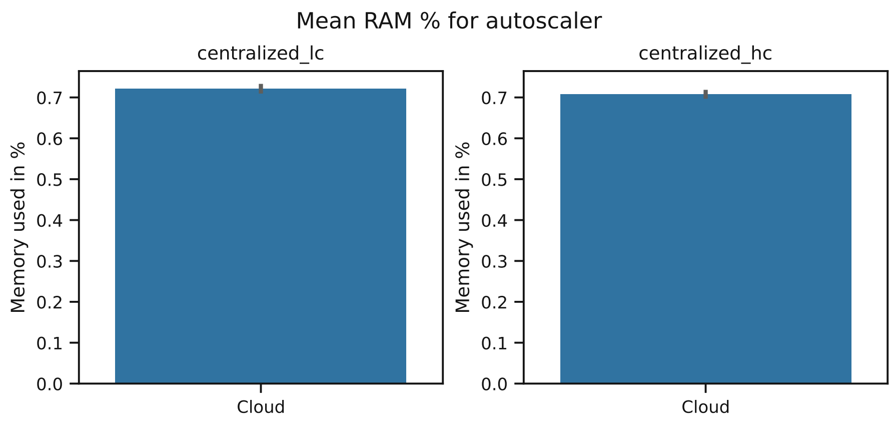


Figure 5.28: Average RAM usage from autoscaler, high and low constraint environment, centralized architecture

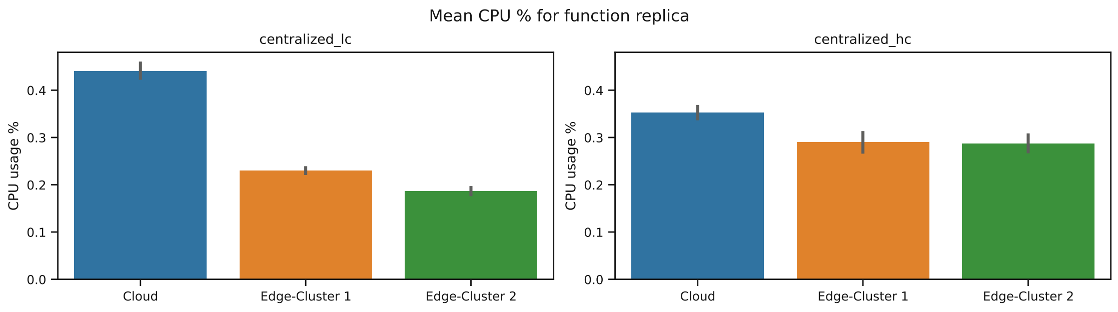


Figure 5.29: Average CPU usage from function replica, high and low constraint environment, centralized architecture

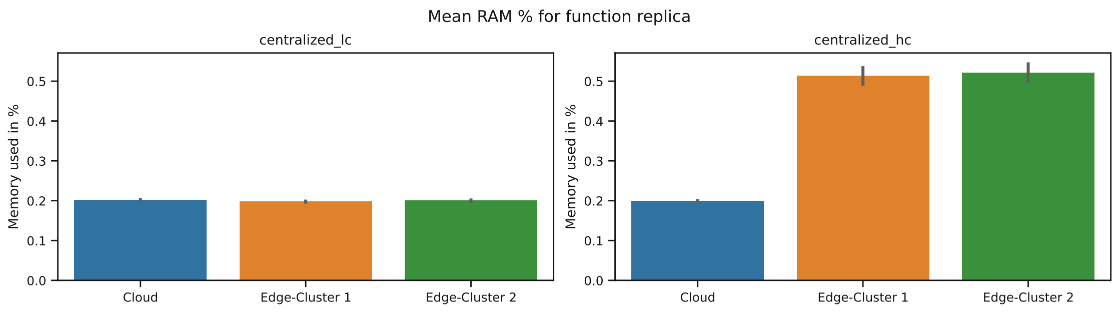


Figure 5.30: Average RAM usage from function replica, high and low constraint environment, centralized architecture

5.3.4 System level *hc_lc* parameter set with decentralized architecture

Figure 5.35 illustrates the average CPU and RAM usage for both constraint environments with a decentralized architecture divided by cluster. We directly see that there is a similar trend between average CPU and average RAM usage. In decentralized_lc, the highest

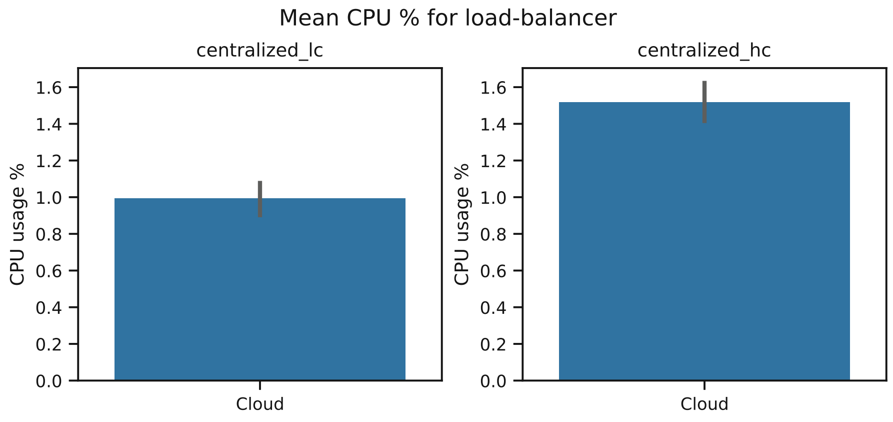


Figure 5.31: Average CPU usage from load-balancer, high and low constraint environment, centralized architecture

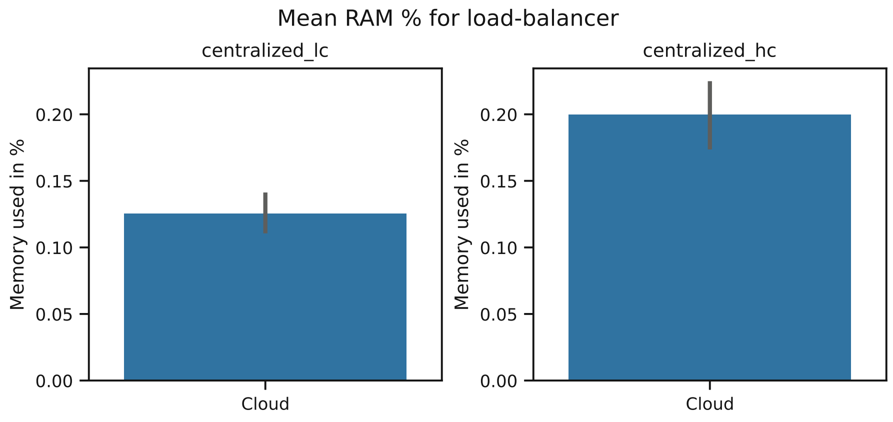


Figure 5.32: Average RAM usage from load-balancer, high and low constraint environment, centralized architecture

mean CPU usage was found in Edge-Cluster 2 with 32.2% and a standard deviation of 23.2%. The lowest one was found in the Cloud with 13.2% and a standard deviation of 14.6%. In decentralized_hc, Edge-Cluster 1 had the highest mean CPU usage of 48% and a standard deviation of 20.5%. The cluster with the lowest mean CPU usage in decentrlalized_hc was the Cloud with 10.5% and a standard deviation of 12.8%.

The same pattern follows for average RAM usage. In decentralized_lc, the highest mean value, 9%, with a standard deviation of 5.3%, was measured in Edge-Cluster 2. The lowest one was measured in the Cloud with 5.4% and a standard deviation of 1.9%. In decentralized_hc, the highest mean RAM usage was in Edge-Cluster 1 with 14.6% and a standard deviation of 5.7%, and the lowest in the Cloud with 5.4% and a standard deviation of 1.8%.

The following figure, Figure 5.36, illustrates the average CPU and RAM usage for each

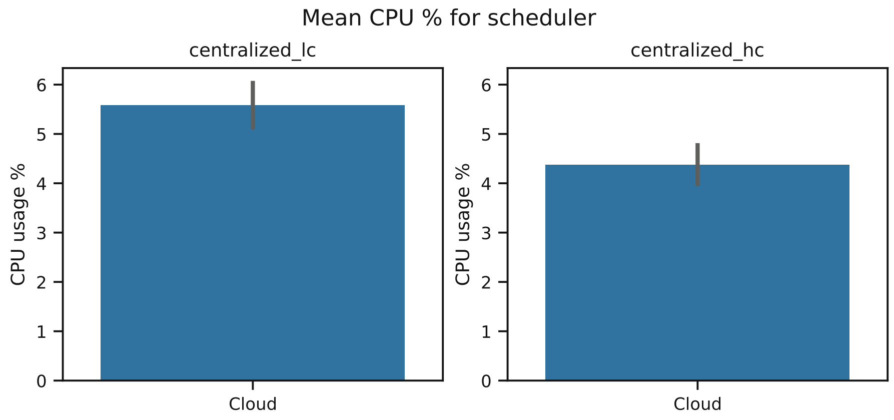


Figure 5.33: Average CPU usage from scheduler, high and low constraint environment, centralized architecture

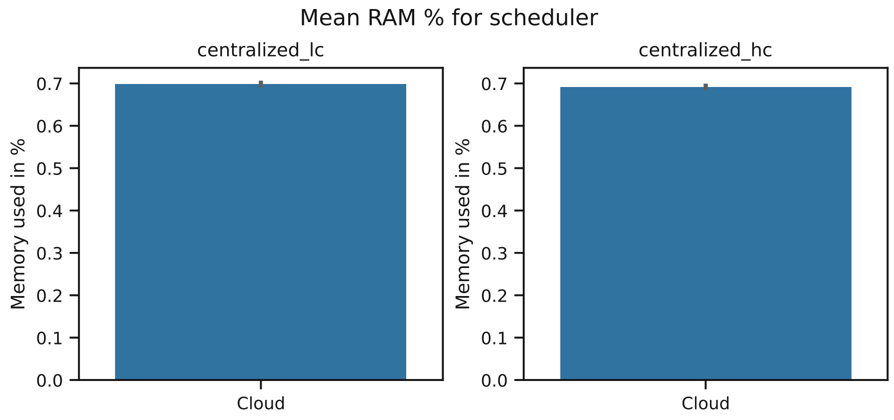


Figure 5.34: Average RAM usage from scheduler, high and low constraint environment, centralized architecture

node type in both decentralized scenarios. Looking at the mean CPU usage of weak nodes, we can see that in decentralized_lc, the usage was 65.2%, with a standard deviation of 9.4%. In decentralized_hc, the mean CPU usage of weak nodes was lower with 61.5% and a standard deviation of 8.5%. The mean CPU usage from strong nodes was also close to each other. In decentralized_hc, the value was at 18.3% with a standard deviation of 14.4%, and in decentralized_lc at 16.6% with a standard deviation of 14.7%.

There is a similar pattern for mean RAM usage for each node type. In decentralized_hc, the value from weak nodes was 18.8% with a standard deviation of 1.5%. Lower, 17.7% with a standard deviation of 0.7%, was the mean RAM usage from weak nodes in decentralized_lc. Also, regarding the strong nodes, the usage in decentralized_hc was higher than in decentralized_lc, with 6.5% with a standard deviation of 1.8% and 5.6% with a standard deviation of 1.7%.

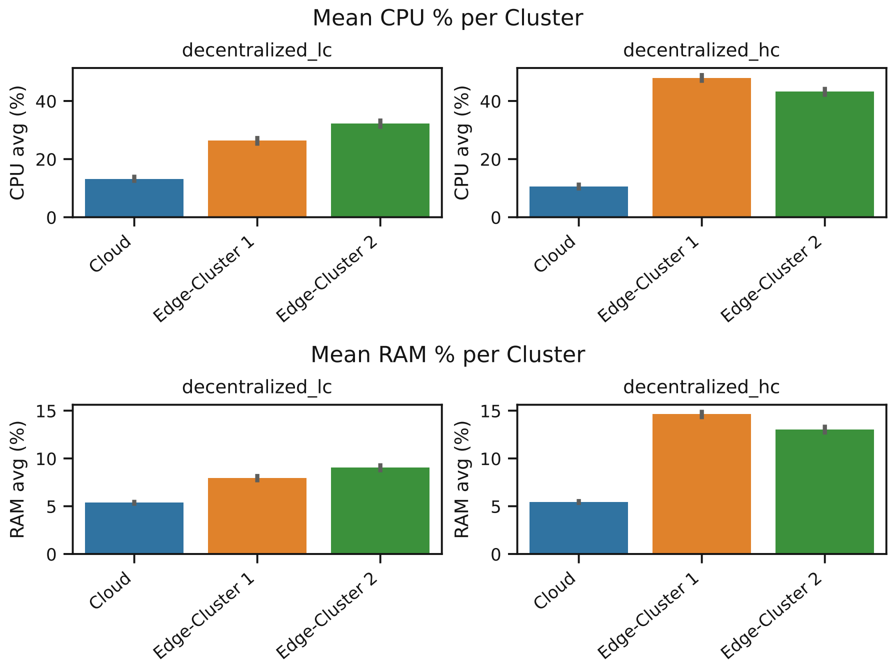


Figure 5.35: Average CPU and RAM usage per Cluster, high and low constraint environment, decentralized architecture

Figure 5.37 depicts the average CPU usage of the autoscaler for each cluster. In decentralized_lc, the highest mean figure was in Edge-Cluster 1 with 1.1%, and a standard deviation of 0.35%, and the lowest one was in the Cloud with 0.98% and a standard deviation of 0.3%. While in decentralized_hc, the same pattern occurs. The mean figures of the clusters were closer to each other. 0.95% with a standard deviation of 0.17% was the highest mean usage value measured in Edge-Cluster 1. The lowest average CPU usage from the autoscaler was measured in the Cloud with 0.92% with a standard deviation of 0.18%.

If we look at Figure 5.38, we see that the average RAM usage for each cluster in decentralized_lc and decentralized_hc were close. The autoscaler in all clusters in both infrastructures had an average RAM usage of around 0.7% and a standard deviation of 0.03%.

Figure 5.39 and Figure 5.40 illustrate a function replica's average CPU and RAM usage. In decentralized_lc, the mean CPU usage was higher in Edge-Cluster 1, with 3.3% and a standard deviation of 3%, than in Edge-Cluster 2. However, Edge-Cluster 2 was lower, with 3.1% and a standard deviation of 2.7%. The situation is flipped in decentralized_hc, where the higher mean usage was measured in Edge-Cluster 2, with 1.5% and a standard deviation of 1.8%, and the lower in Edge-Cluster 1, with 1.45% and a standard deviation of 1.7%.

In decentralized_lc, the highest mean RAM usage was measured in Edge-Cluster 2 with

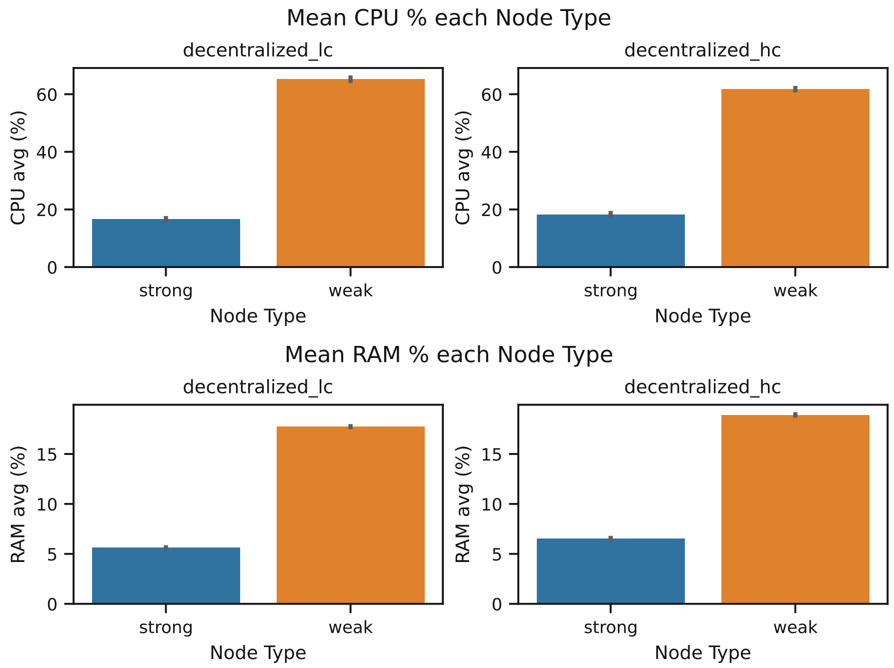


Figure 5.36: Average CPU and RAM usage per Node Type, high and low constraint environment, decentralized architecture

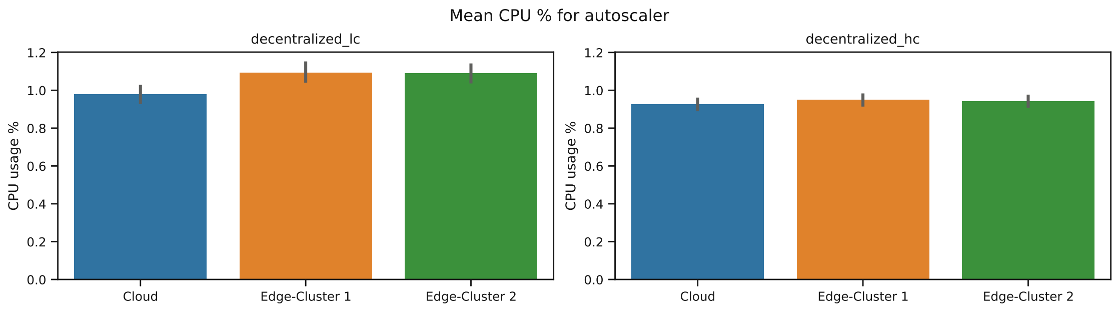


Figure 5.37: Average CPU usage from autoscaler, high and low constraint environment, decentralized architecture

0.51% and a standard deviation of 0.23%, and the lowest in Edge-Cluster 1 with 0.42% and a standard deviation of 0.27%. In decentralized_hc, it is the other way around. There, the highest mean RAM usage was found in Edge-Cluster 1 with 0.64% and a standard deviation of 0.08%. Function replica in Edge-Cluster 2 had a mean RAM usage of 0.51% with a standard deviation of 0.07%.

Looking at Figure 5.41 and Figure 5.42, we see that both mean CPU and RAM usage of the load-balancers in Edge-Cluster 1 and Edge-Cluster 2 were nearly identical in the respective infrastructure. In decentralized_lc, the mean CPU usage was around 1.2% with a standard deviation of 1.96%, and in decentralized_hc, 1.15% with a standard

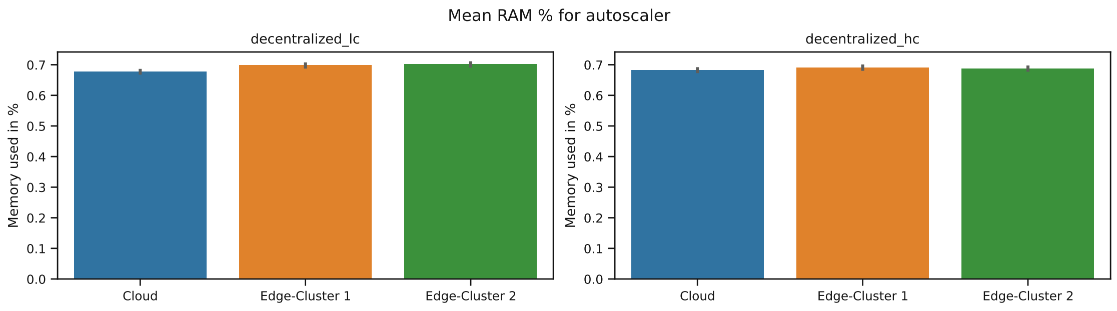


Figure 5.38: Average RAM usage from autoscaler, high and low constraint environment, decentralized architecture

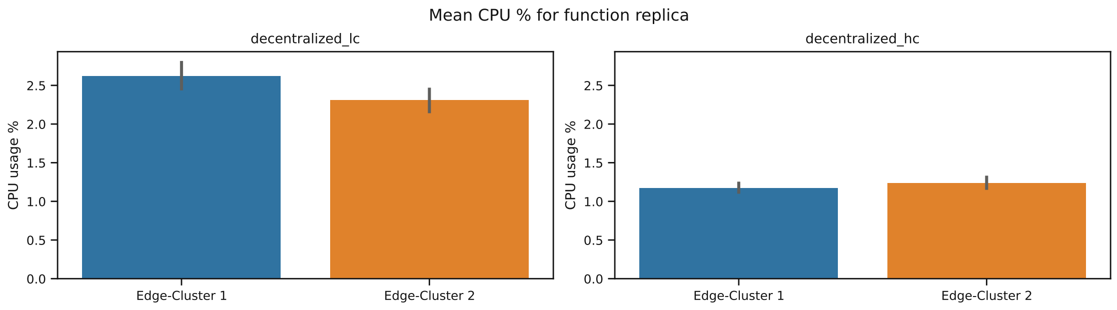


Figure 5.39: Average CPU usage from function replica, high and low constraint environment, decentralized architecture

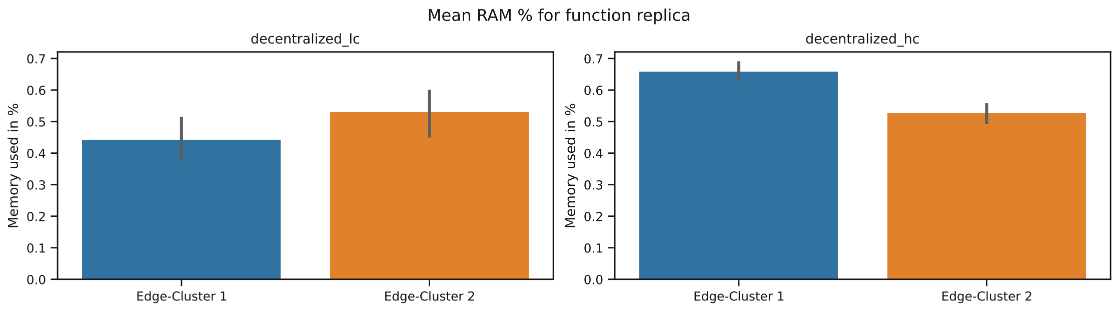


Figure 5.40: Average RAM usage from function replica, high and low constraint environment, decentralized architecture

deviation of 1.1%. In both infrastructures, the average CPU usage of the load-balancer in the Cloud was close to 0%.

In decentralized_lc, the highest mean RAM usage was in Edge-Cluster 2, with a figure of 0.26% and a standard deviation of 0.27%, while in decentralized_hc, the highest mean value was in Edge-Cluster 1, with 0.14% and a standard deviation of 0.11%. For both infrastructures, the lowest was in the Cloud with 0.055% and a standard deviation of nearly 0%.

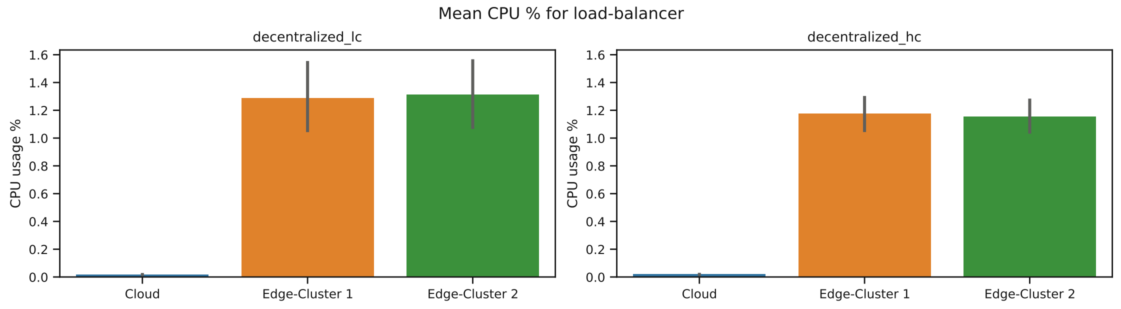


Figure 5.41: Average CPU usage from load-balancer, high and low constraint environment, decentralized architecture

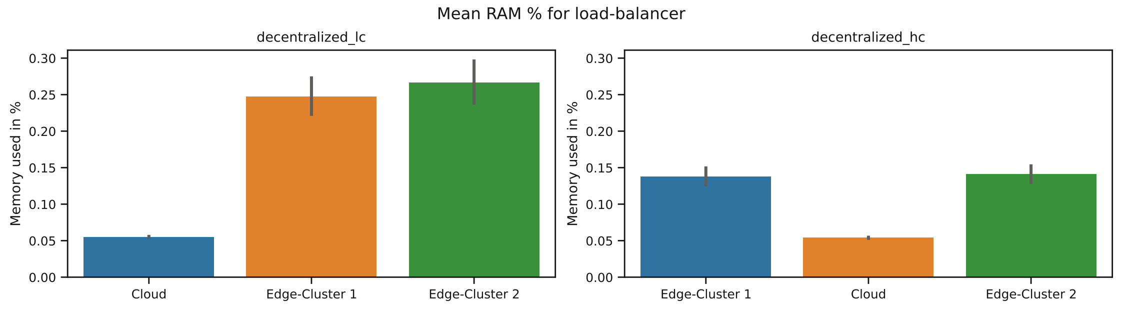


Figure 5.42: Average RAM usage from load-balancer, high and low constraint environment, decentralized architecture

Looking at Figure 5.43, we see a noticeable difference in mean CPU usage from a scheduler between the two environments. In decentralized_lc, the CPU mean usage was nearly identical in all clusters, with a value of around 4% and a standard deviation of 2%. In contrast, in decentralized_hc, all clusters had a mean CPU usage of around 2% with a standard deviation of around 2.3%.

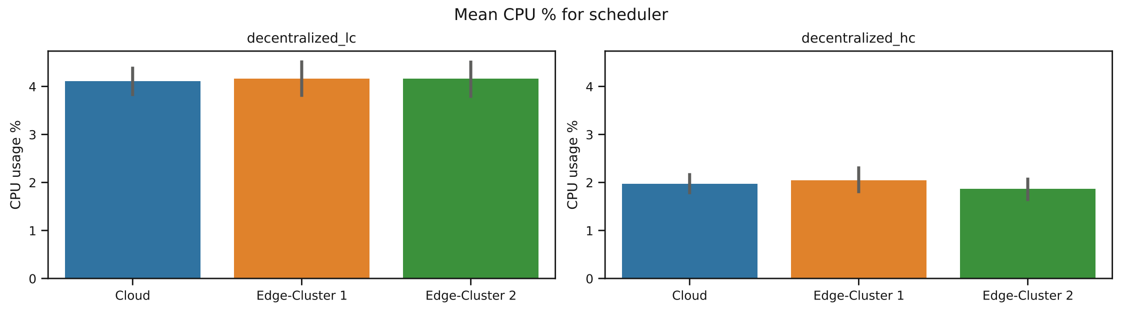


Figure 5.43: Average CPU usage from scheduler, high and low constraint environment, decentralized architecture

From Figure 5.44, we see that the schedulers in each cluster in decentralized_lc and decentralized_hc had the same mean RAM usage of around 0.69%. The standard deviation was at 0.03%.

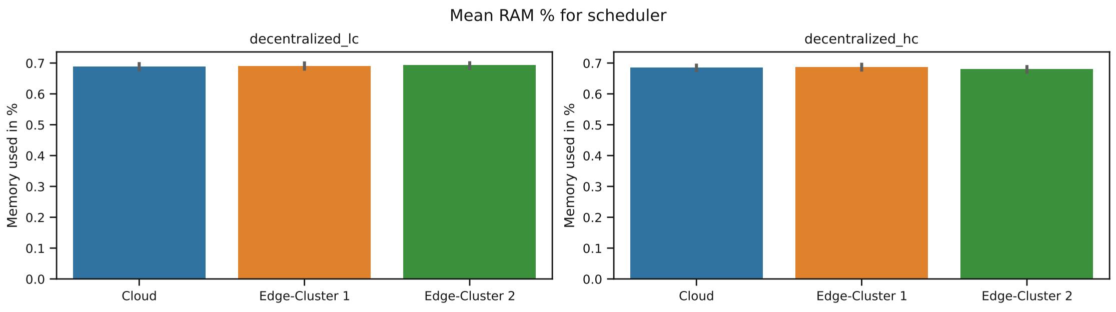


Figure 5.44: Average RAM usage from scheduler, high and low constraint environment, decentralized architecture

5.3.5 System level *hc_lc* parameter set with distributed architecture

Figure 5.45 illustrates each cluster's average CPU and RAM usage in both infrastructures with a distributed architecture. In distributed_lc, the mean CPU usage for Edge-Cluster 3 and 2 was very similar and the highest with a mean value of around 31.3% and a standard deviation of 24%. The lowest one was in Edge-Cluster 1 with 25.6% and a standard deviation of 23.7%. However, in distributed_hc, the highest mean value was measured in Edge-Cluster 1 with a figure of 55% and a standard deviation of 22.4%, and the lowest mean value, 44.6% with a standard deviation of 20.2%, was found in Edge-Cluster 3.

There is a similar pattern for average RAM usage. In distributed_lc Edge-Cluster 3 and 2 also have a very similar and highest mean usage with around 9.5% and a standard deviation of 5.6%. The lowest mean RAM usage was in Edge-Cluster 1 with 8.3% and a standard deviation of 5.5%. Regarding distributed_hc, the highest mean usage was in Edge-Cluster 1 with 15.9%, and a standard deviation of 6.7%, and the lowest one was in Edge-Cluster 3 with a figure of 12.6% and a standard deviation of 4.9%.

The figure, Figure 5.46, depicts the average CPU and RAM usage for weak and strong nodes for the distributed architecture in both infrastructures. When we look at the mean CPU usage of weak nodes in distributed_hc, the figure was higher, with 67.5% and a standard deviation of 10.4%, than in distributed_lc, with 63.3% and a standard deviation of 9.7%. A similar situation with strong nodes, where the higher mean value was measured in distributed_hc, with a value of 30% and a standard deviation of 13.3%, and the lower in distributed_lc, with 17.1% and a standard deviation of 14%.

Not only in mean CPU usage but also in mean RAM usage, distributed_hc had a higher usage than distributed_lc in strong and weak nodes. The mean usage in weak nodes was 19.8% with a standard deviation of 2.4%, and in strong nodes was 7.8% with a standard deviation of 0.8%, in distributed_hc. In the low constraint infrastructure, the mean

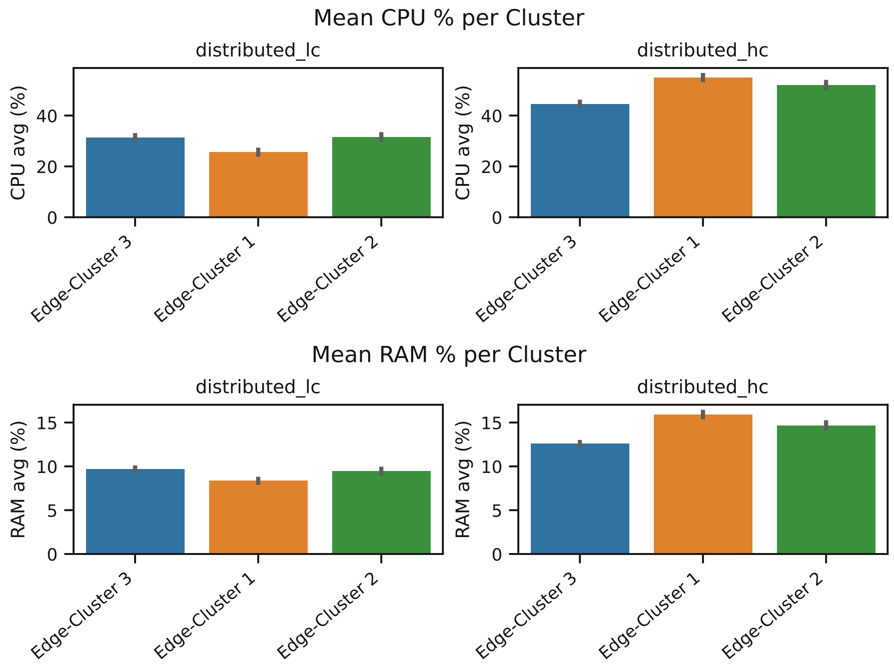


Figure 5.45: Average CPU and RAM usage per Cluster, high and low constraint environment, distributed architecture

RAM usage in weak nodes was at 18.1% with a standard deviation of 0.9%, and in strong nodes, 5.9% with a standard deviation of 1.5%.

The following two figures, Figure 5.47 and Figure 5.48 illustrate the average CPU and RAM usage of the autoscalers in the clusters for both environments. The figures suggest that the mean RAM and CPU usage was nearly identical for each infrastructure. In distributed_lc, the average CPU usage for autoscalers in all three clusters was around 1.1% with a standard deviation of 0.57%, and the mean RAM usage for all was at 0.7% with a standard deviation of 0.04%. While the mean RAM usage in distributed_lc for all clusters is similar to distributed_lc, the mean CPU usage of the autoscale for the three clusters is higher, with a value of around 1.72% and a standard deviation of around 0.5%.

Figure 5.49 and Figure 5.50 illustrate the average CPU and RAM usage of a function replica on a cluster for both distributed_lc and distributed_hc. Looking at Figure 5.49, we see that the mean CPU usage of both Edge-Cluster 1 and 2 in distributed_lc is nearly identical, with a figure of 4.7%, where they had a standard deviation of 7.4%. The same case is in distributed_hc, where function replica in Edge-Cluster 1 and 2 have a near identical mean CPU usage of 1.8% with a standard deviation of 3%. In both environments, Edge-Cluster 3 had the lowest mean figure, with a value of 3.3% and a standard deviation of 2.5& in distributed_lc, and 0.7% with a standard deviation of 0.6% in distributed_hc.

In distributed_lc, the mean RAM usage of a function replica was nearly identical in

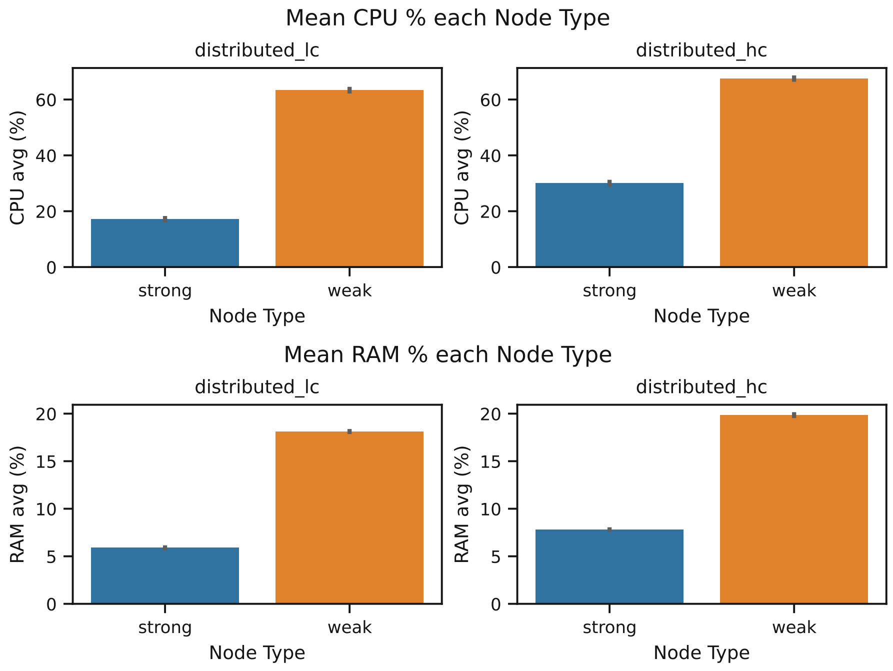


Figure 5.46: Average CPU and RAM usage per Node Type, high and low constraint environment, distributed architecture

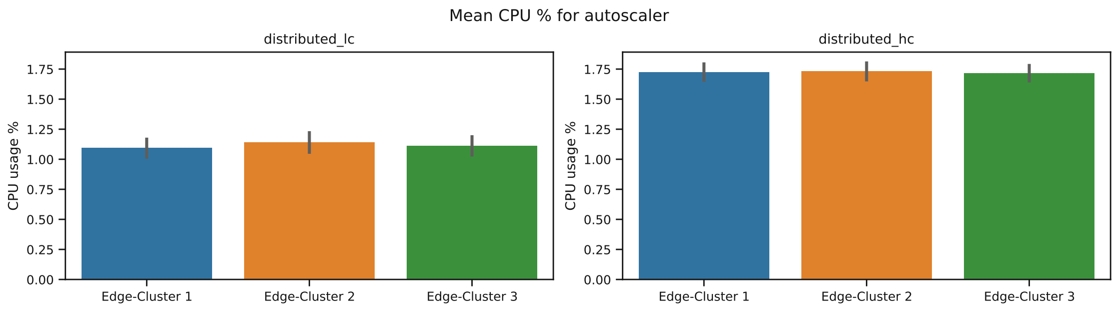


Figure 5.47: Average CPU usage from autoscaler, high and low constraint environment, distributed architecture

Edge-Cluster 1 and 2 with a value of 0.6% and a standard deviation of 0.15%. The mean RAM usage in Edge-Cluster 3 was only 0.21% with a standard deviation of 0.02%. Similarly, in distributed_hc, the lowest average RAM usage figure was measured in Edge-Cluster 3, with 0.2% and a standard deviation of 0.025%. The highest mean value was in Edge-Cluster 1 with 0.77% and a standard deviation of 0.12%.

Figure 5.51 depicts the average CPU usage of the load-balancer in a cluster for both distributed_lc and distributed_hc. In distributed_lc, the load-balancer in each cluster had a similar mean CPU usage of 0.4% with a standard deviation of 0.3%. In distributed_hc, the mean CPU usage of the load-balancers in each cluster was also close. The highest

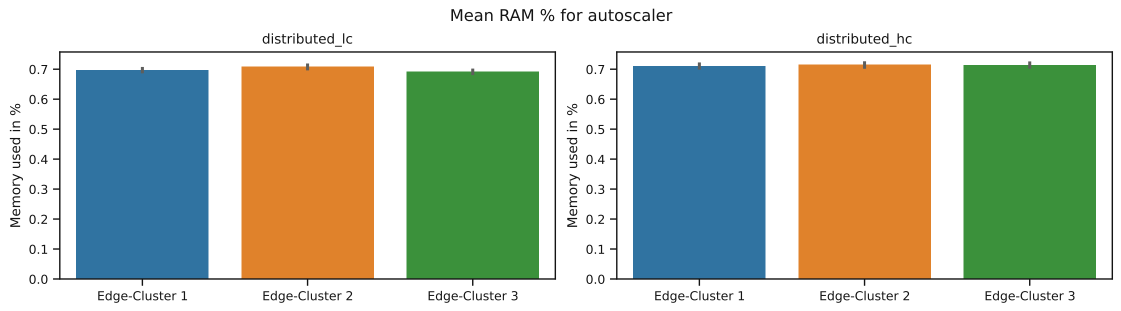


Figure 5.48: Average RAM usage from autoscaler, high and low constraint environment, distributed architecture

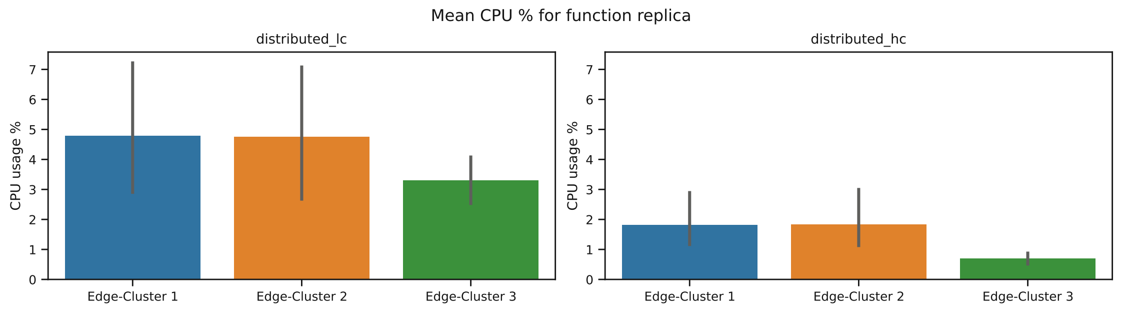


Figure 5.49: Average CPU usage from function replica, high and low constraint environment, distributed architecture

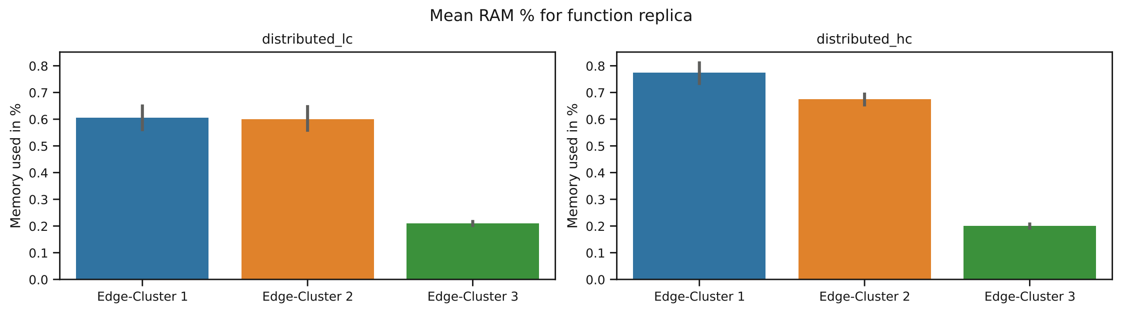


Figure 5.50: Average RAM usage from function replica, high and low constraint environment, distributed architecture

mean value was in Edge-Cluster 2 with 0.94% and a standard deviation of 0.9%, and the lowest in Edge-Cluster 3 with 0.89% and a standard deviation of 0.85%.

Looking at Figure 5.52, we see that the mean RAM usage of the load-balancer in each cluster in their infrastructure was nearly identical. In distributed_lc, the mean RAM usage in each cluster was at 0.08% with a standard deviation of 0.015%, and in distributed_hc, 0.09% with a standard deviation of 0.035%.

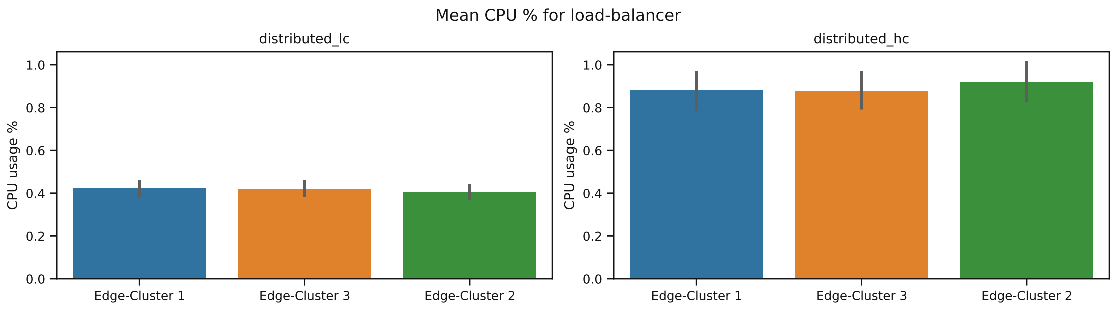


Figure 5.51: Average CPU usage from load-balancer, high and low constraint environment, distributed architecture

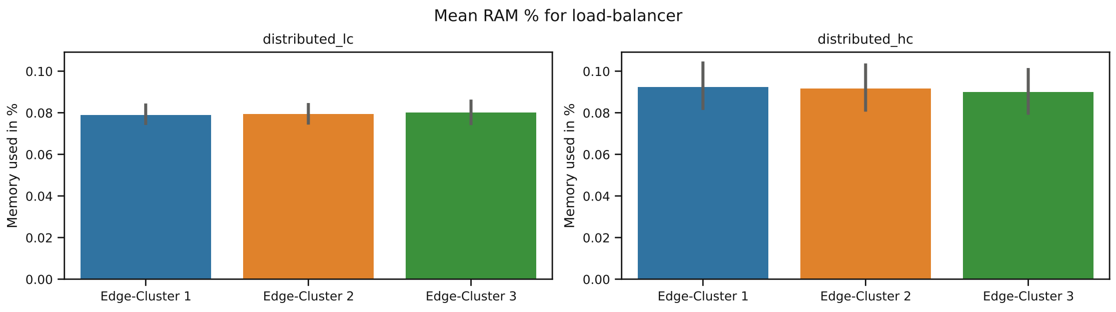


Figure 5.52: Average RAM usage from load-balancer, high and low constraint environment, distributed architecture

The average CPU usage of a scheduler in a cluster for both distributed_hc and distributed_lc is illustrated in Figure 5.53. In distributed_lc, the lowest mean CPU usage was in Edge-Cluster 3 with 2.4% and a standard deviation of 2.9%, and the highest in Edge-Cluster 2 with 2.8% and a standard deviation of 3.2%. Similarly, the highest in distributed_hc was also Edge-Cluster 2 with 9.3% with a standard deviation of 2.3%, and the lowest was also Edge-Cluster 3 with 3.7% with a standard deviation of 2.8%.

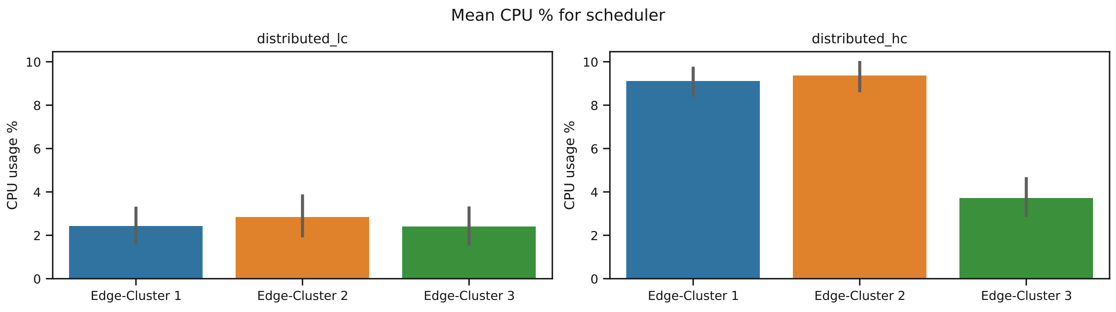


Figure 5.53: Average CPU usage from scheduler, high and low constraint environment, distributed architecture

Figure 5.54 represents the average RAM usage of a scheduler in each cluster for distributed_hc and distributed_lc. We see that in all clusters for both infrastructures, the mean RAM usage was around 0.7%. The standard deviation was around 0.7%.

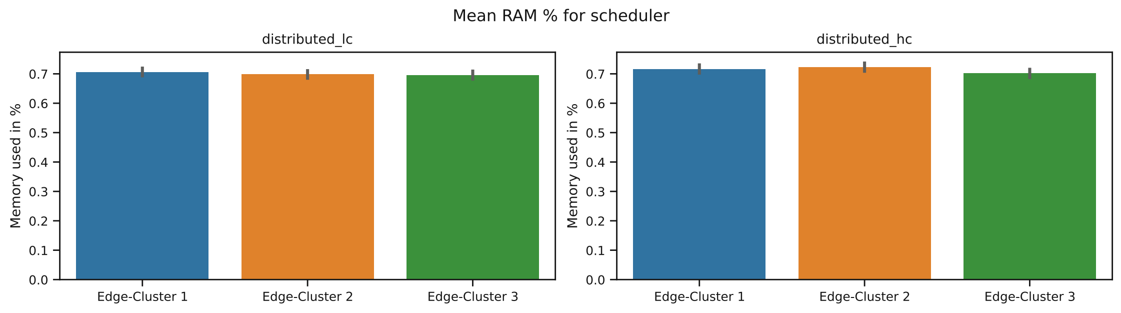


Figure 5.54: Average RAM usage from scheduler, high and low constraint environment, distributed architecture

5.3.6 Network level *hc* parameter set

Figure 5.55 illustrates the distribution of network throughput per second for each architecture in high constraint infrastructures in box plots. The quartiles (q1, q2, q3) for centralized_hc were: (0.1MB/s, 0.2MB/s, 2.4MB/s), and its whiskers (whisklo, whiskhi) were: (0.0MB/s, 5.9MB/s). For decentralized_hc, we measured similar values where the quartiles were (0.0MB/s, 0.1MB/s, 3.1MB/s), and the whiskers (0.0MB/s, 7.7MB/s). The biggest box was in distributed_hc with (0.1MB/s, 0.4MB/s, 4.7MB/s) as quartiles and (0.0MB/s, 11.7MB/s) as its whiskers.

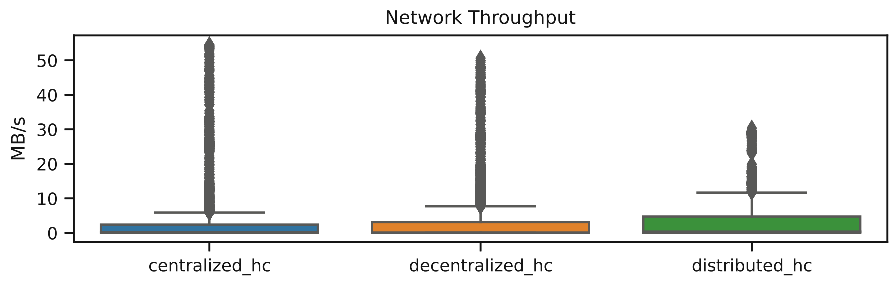


Figure 5.55: Distribution of Network Throughput per second per infrastructure, high constraint infrastructure, all architectures

The following figure, Figure 5.56, depicts the distribution of latency values in milliseconds through box plots. Each experiment's latency values are further categorized by the start and end cluster from the user request path. For decentralized_hc, we have four different paths. The quantiles one, two, and three (q1, q2, q3) for these paths are (from left to right): (59.1ms, 90.4ms, 453.8ms), (116.1ms, 198.2ms, 535.4ms), (59.2ms, 90.9ms, 457.6ms) and (112.6ms, 166.7ms, 479.5ms). The low and high whiskers (whisklo, whiskhi) are (from left to right): (51.7ms, 1045.7ms), (99.0ms, 1151.6ms), (51.4ms, 1052.5ms) and (98.3ms, 1026.2ms). Distributed_hc features a lot of lower values. The quantiles from left to right are: (57.6ms, 72.9ms, 444.8ms), (57.8ms, 73.6ms, 432.6ms) and (57.6ms, 73.9ms, 461.3ms), and the whiskers from left to right are: (51.7ms, 1022.8ms), (51.5ms, 990.6ms) and (51.1ms, 1062.4ms). In centralized_hc, we have the most number of paths. The quantiles for each path from left to right are: (162.8ms, 218.7ms, 408.6ms), (217.5ms, 281.4ms, 481.3ms), (218.1ms, 274.5ms, 468.9ms), (163.5ms, 208.5ms, 409.4ms), (218.1ms, 284.2ms, 474.2ms) and (217.5ms, 277.7ms, 457.3ms). For the low and high whiskers, the following values were able to be collected (from left to right): (155.4ms, 777.2ms), (208.5ms, 875.2ms), (208.9ms, 841.8ms), (155.8ms, 778.3ms), (209.0ms, 854.6ms) and (208.7ms, 798.3ms).

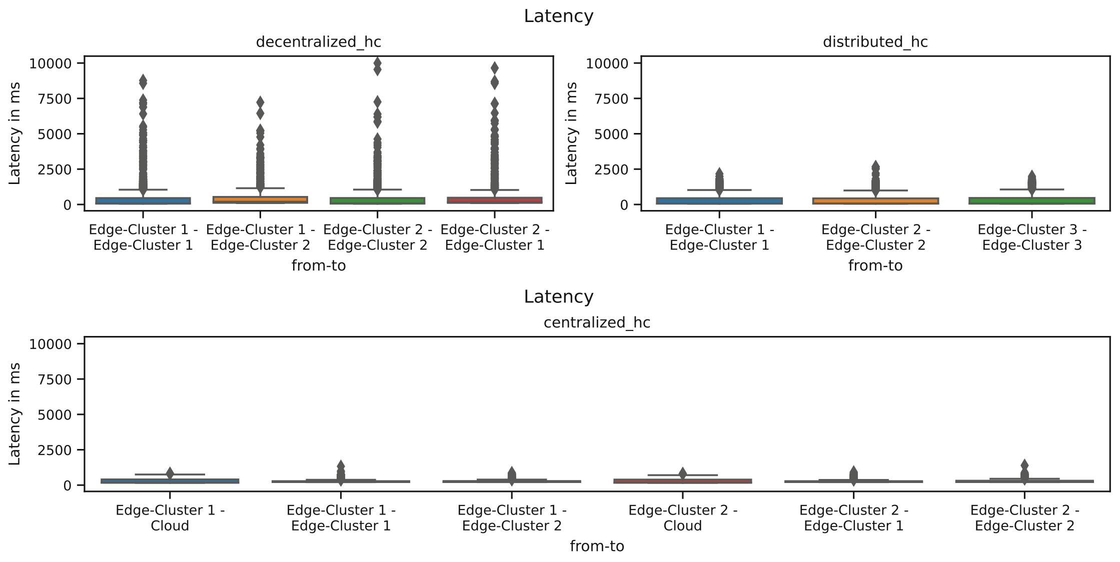


Figure 5.56: Distribution of latency values for each request path per architecture, high constraint infrastructure, all architectures

Figure 5.57 describes the average net throughput of the autoscaler categorized by the cluster for each architecture in a high constraint environment. In centralized_hc, the mean usage was 0.2 MB/s with a standard deviation of 0.03 MB/s. In decentralized_hc, all clusters had a similar mean of around 0.17 MB/s with a standard deviation of around 0.2 MB/s. Regarding distributed_hc, the highest mean value was measured in Edge-Cluster 1 with 0.26 MB/s and a standard deviation of 0.5 MB/s, and the lowest in Edge-Cluster 3 with 0.25 MB/s and a standard deviation of 0.05 MB/s.

The following figure, Figure 5.58 illustrates the average net throughput of a function replica. The highest mean value was 0.45 MB/s with a standard deviation of 0.9 MB/s in the Cloud in centralized_hc. The lowest was 0.27 MB/s, with a standard deviation of 0.74 MB/s in Edge-Cluster 1. In decetralized_hc, the higher mean value was in Edge-Cluster 2 with 1.1 MB/s and a standard deviation of 1.8 MB/s, while the lower average figure was in Edge-Cluster 1 with 0.9 MB/s and a standard deviation of 1.5 MB/s. In distributed_hc, the highest measured mean value was in Edge-Cluster 2 with

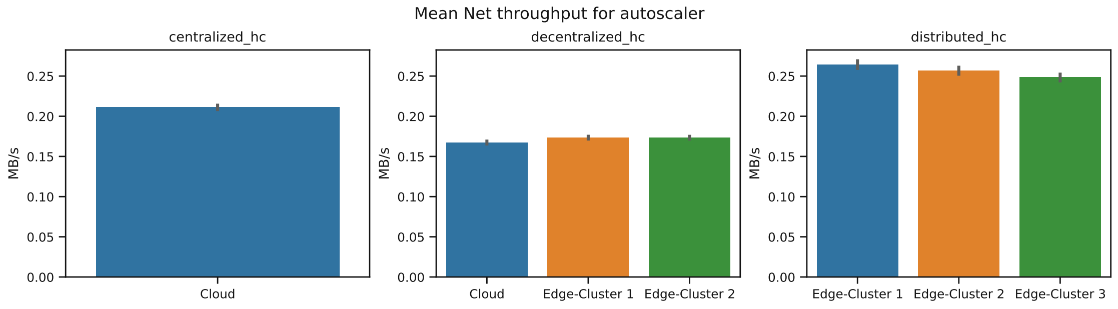


Figure 5.57: Average Net throughput of autoscaler high constraint infrastructure, all architectures

0.83 MB/s and a standard deviation of 1.6 MB/s, and the lowest one was in Edge-Cluster 3 with 0.6 MB/s and a standard deviation of 1.3 MB/s.

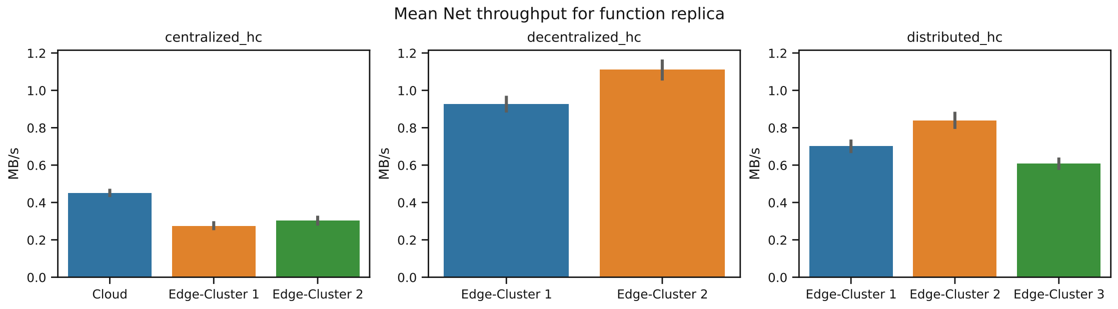


Figure 5.58: Average Net throughput of function replica, high constraint infrastructure, all architectures

Figure 5.59 depicts the average net throughput of a load-balancer for each architecture in a high constraint environment. In centralized_hc, the mean usage was 32.4 MB/s with a standard deviation of 18.68 MB/s. The highest mean value in decentralized_hc was in Edge-Cluster 1 with 19.7 MB/s and a standard deviation of 14.3 MB/s. The lowest was in the Cloud, with 2 MB/s and a standard deviation of 0.28 MB/s. In distributed_hc, all load-balancers had a nearly identical usage of around 18 MB/s with a standard deviation of around 11.7 MB/s.

The average net throughput of a scheduler for each architecture in a high constraint environment is illustrated in Figure 5.60. The average net throughput of the scheduler in centralized_hc was 0.23 MB/s with a standard deviation of 0.02 MB/s. The schedulers in the clusters in decentralized_hc all had a similar net throughput of around 0.18 MB/s with a standard deviation of 0.02 MB/s. In distributed_hc, the highest mean net throughput was measured in Edge-Cluster 2 with 0.31 MB/s and a standard deviation of 0.06 MB/s. The lowest average value was Edge-Cluster-3, with 0.29 MB/s and a standard deviation of 0.05 MB/s.

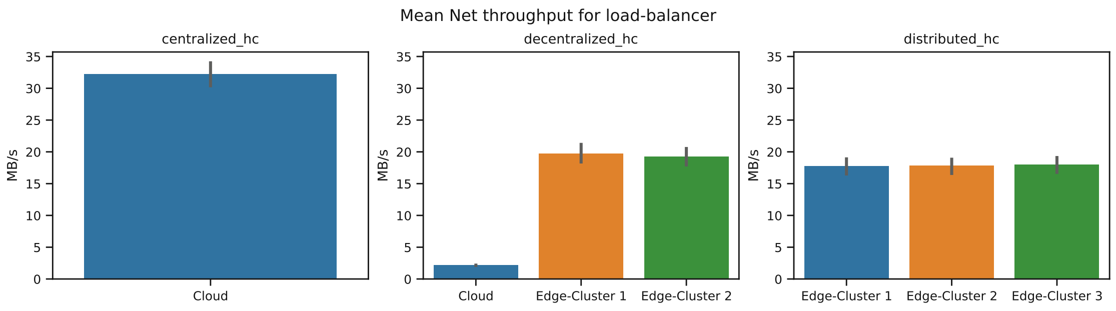


Figure 5.59: Average Net throughput of load-balancer, high constraint infrastructure, all architectures

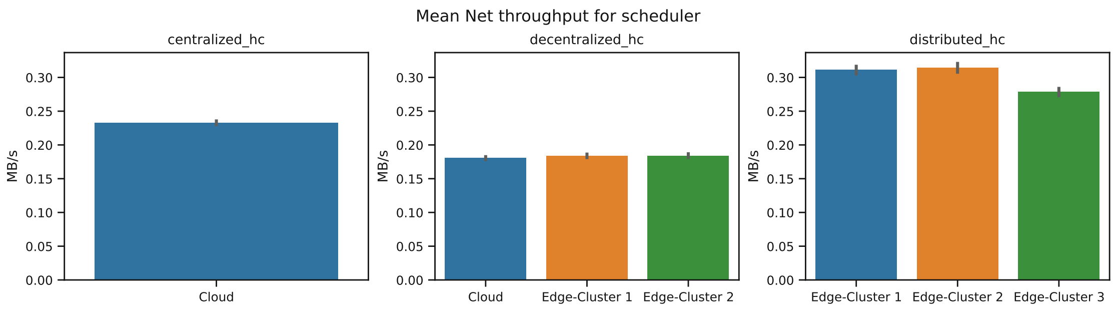


Figure 5.60: Average Net throughput of scheduler, high constraint infrastructure, all architectures

5.3.7 Network level *lc* parameter set

Figure 5.61 depicts the distribution of network throughput per second for each architecture in a low constraint infrastructure. In centralized_lc, we calculated the following quartiles (q1, q2, q3): (0.1MB/s, 0.2MB/s, 1.5MB/s), and following whiskers (whisklo, whiskhi): (0.0MB/s, 3.7MB/s). Decentralized_hc had a slightly bigger range with its quartiles (0.1MB/s, 0.1MB/s, 2.8MB/s) and whiskers (0.0MB/s, 6.9MB/s). An even bigger range was measured in distributed_lc with (0.0MB/s, 0.1MB/s, 3.8MB/s) for its quartiles and (0.0MB/s, 9.3MB/s) for its whiskers.

The box plots in Figure 5.62 illustrate the distribution of latency values in milliseconds and follow the same layout as Figure 5.56. For the four paths in decentralized_lc, we found the following (from left to right) quartile values (q1, q2, q3): (41.9ms, 45.8ms, 55.6ms), (79.4ms, 91.3ms, 120.6ms), (41.9ms, 45.5ms, 55.0ms) and (81.1ms, 96.9ms, 123.9ms). Regarding the low and high whiskers (whisklo, whiskhi) in decentralized_lc, we calculated the following values (from left to right): (36.6ms, 76.2ms), (68.1ms, 180.7ms), (36.5ms, 74.6ms) and (67.5ms, 184.8ms). In distributed_lc, the quartiles and whiskers were nearly identical across the request path. From left to right the following quartiles (q1,q2,q3) : (41.9ms, 43.6ms, 49.2ms), (41.4ms, 43.1ms, 47.4ms) and (41.8ms, 43.6ms, 49.0ms), and following whiskers (whisklo, whiskhi): (36.2ms, 60.2ms), (36.2ms, 56.3ms)

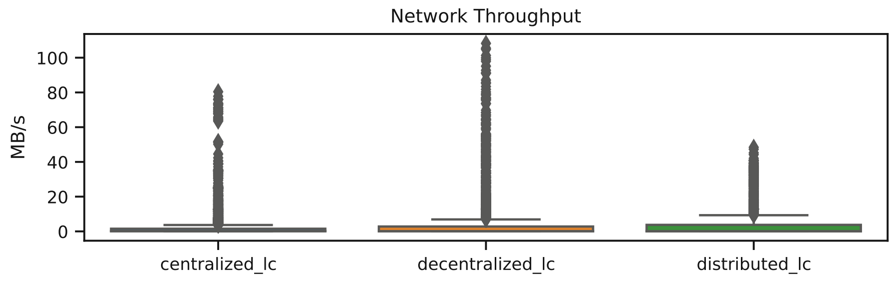


Figure 5.61: Distribution Network Throughput per second per infrastructure, low constraint infrastructure, all architectures

and (36.5ms, 59.8ms) were found. In centralized_hc the quartiles (q1, q2, q3) for the six paths were as follows (from left to right): (111.3ms, 113.0ms, 115.9ms), (146.5ms, 147.9ms, 150.6ms), (146.4ms, 148.0ms, 150.9ms), (111.4ms, 113.1ms, 116.3ms), (146.6ms, 148.1ms, 151.2ms) and (146.4ms, 148.0ms, 150.9ms). The whiskers for centralized_hc are as follows (from left to right): (106.3ms, 122.8ms), (141.4ms, 156.8ms), (141.8ms, 157.6ms), (106.3ms, 123.7ms), (142.3ms, 158.0ms) and (141.8ms, 157.7ms).

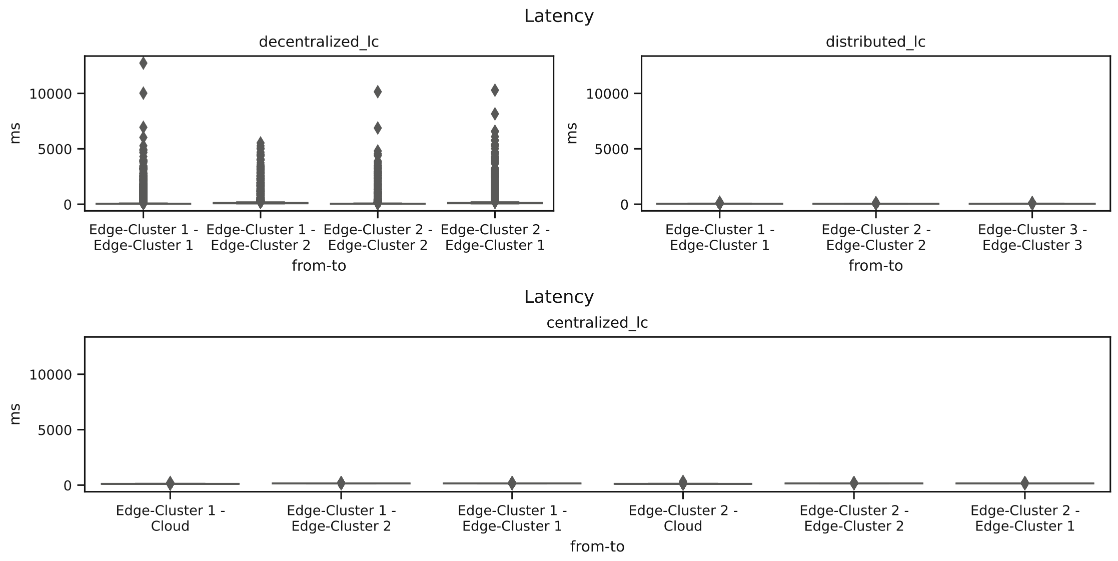


Figure 5.62: Distribution of latency values for each request path per architecture, low constraint environment, all architectures

In Figure 5.63, we see each cluster's average net throughput of an autoscaler of each architecture in a low constraint infrastructure. In centralized_lc, the mean value was 0.2 MB/s with a standard deviation of 0.03 MB/s. In decentralized_lc, the mean net throughput was for each cluster nearly identical. In each cluster, the autoscaler had a mean net throughput of around 0.16 MB/s and a standard deviation of around 0.03 MB/s. Similarly, the autoscalers in the clusters in distributed_lc also had a nearly equal net throughput of around 0.18 MB/s with a standard deviation of 0.05 MB/s.

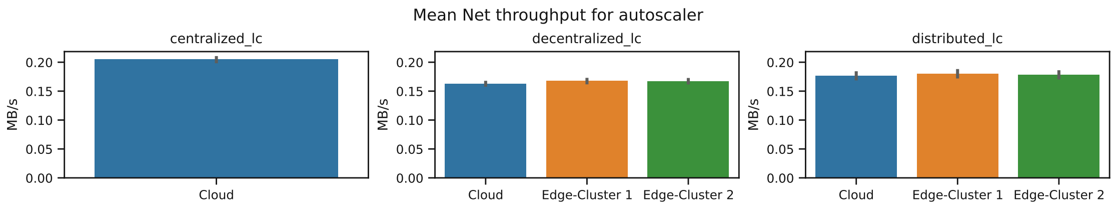


Figure 5.63: Average Net throughput of autoscaler low constraint infrastructure, all architectures

Next, we have Figure 5.64, which describes the average net throughput of a function replica of each cluster in each architecture in a low constraint infrastructure. In centralized_lc, the highest mean net throughput was in the Cloud with 0.5 MB/s with a standard deviation of 0.97 MB/s. The lowest cluster was Edge-Cluster 1, with 0.22 MB/s and a standard deviation of 0.38 MB/s. In decentralized_lc, the higher measured mean net throughput was in Edge-Cluster 2 with 3.2 MB/s and a standard deviation of 5.2 MB/s, and the lower one in Edge-Cluster 1 with 3 MB/s and a standard deviation of 4.9 MB/s. For distributed_lc, we see that the highest measured mean net throughput was in Edge-Cluster 3 with 2.2 MB/s and a standard deviation of 3.5 MB/s. Edge-Cluster 2's lowest mean throughput was measured at 1.8 MB/s and a standard deviation of 3.2 MB/s.

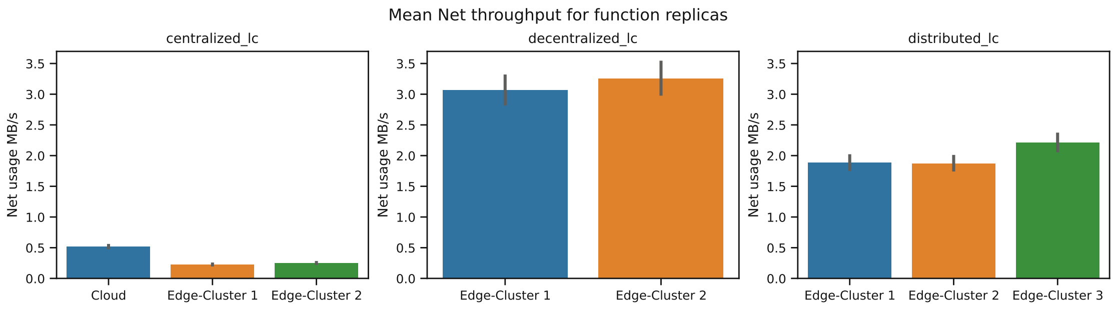


Figure 5.64: Average Net throughput of function replica, low constraint infrastructure, all architectures

Figure 5.65 illustrates each cluster's average net throughput of a load-balancer using each architecture in low constraint infrastructure. In centralized_lc, the mean net throughput of the load-balancer was 25.8 MB/s with a standard deviation of 23.6 MB/s. In contrast, in decentralized_lc, the Cloud had the lowest mean net throughput of 2.4 MB/s with a standard deviation of 0.57 MB/s. The highest was measured in Edge-Cluster 2 with 25.6 MB/s and a standard deviation of 25.1 MB/s. In distributed_lc, the values across the clusters are relatively close. The highest mean net throughput was in Edge-Cluster 1, with 16.6 MB/s and a standard deviation of 13 MB/s, and the lowest was in Edge-Cluster 3, with 15.8 MB/s and a standard deviation of 12.4 MB/s.

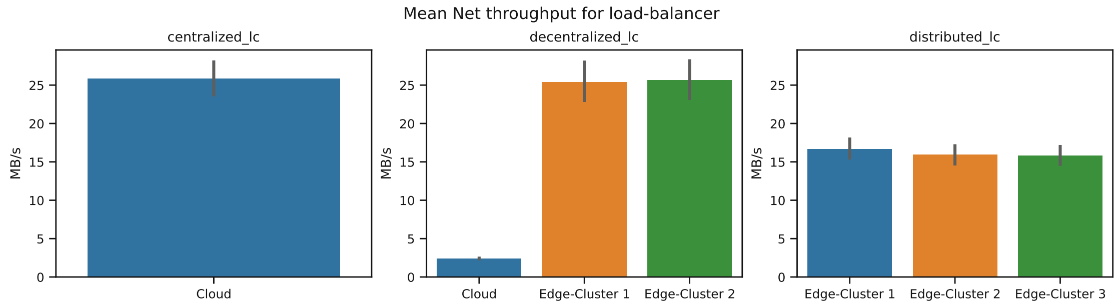


Figure 5.65: Average Net throughput of load-balancer, low constraint infrastructure, all architectures

With Figure 5.66, we depict the average network throughput of a scheduler for each cluster using all three architectures in low constraint infrastructures. In centralized_lc, we had a mean value of 0.24 MB/s with a standard deviation of 0.04 MB/s. For decentralized_lc, the average network throughput for each cluster was relatively close to each other. All clusters had a mean value of around 0.22 MB/s with a standard deviation of around 0.06 MB/s. We have a similar case in distributed_lc, where the schedulers in each cluster had a mean network throughput of around 0.2 MB/s with a standard deviation of around 0.09 MB/s.

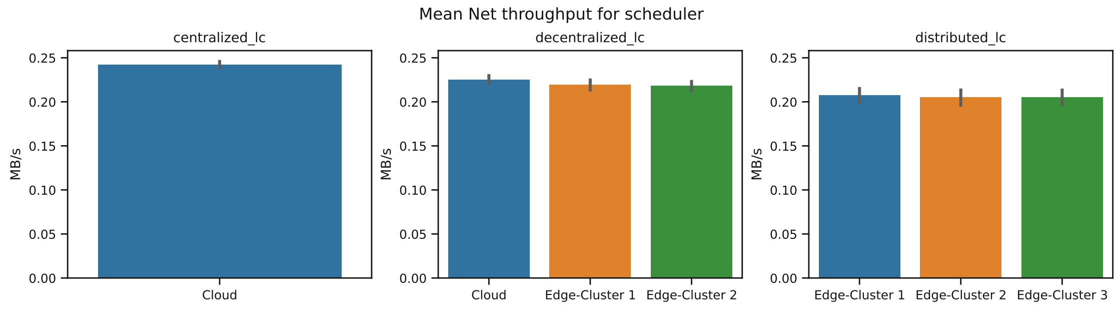


Figure 5.66: Average Net throughput of scheduler, low constraint infrastructure, all architectures

5.3.8 Network level *hc_lc* parameter set with centralized architecture

Figure 5.67 describes the distribution of network throughput per second of both environments with a centralized architecture. The quartiles (q1, q2, q3) of centralized_lc were at (0.1MB/s, 0.2MB/s, 2.0MB/s), and its whiskers (whisklo, whiskhi) were at (0.0MB/s, 4.9MB/s). Slightly smaller, in centralized_hc the quartiles were at (0.1MB/s, 0.2MB/s, 1.7MB/s), and the whiskers were at (0.0MB/s, 4.1MB/s).

In Figure 5.68, the distribution of latency values for user requests path for low and high constraint environments with centralized architecture is illustrated. For centralized_lc and centralized_hc, we had six different request paths. In centralized_lc the following quartiles

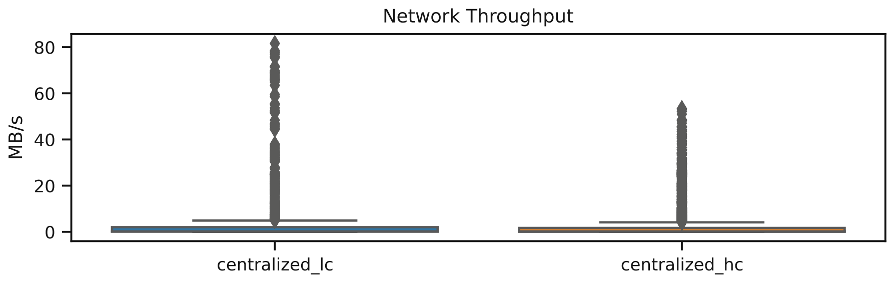


Figure 5.67: Distribution Network Throughput per second per infrastructure, high and low constraint environment, centralized architecture

(q1,q2,q3) were calculated (from left to right): (111.3ms, 112.9ms, 115.8ms), (146.3ms, 147.9ms, 150.7ms), (146.4ms, 148.0ms, 150.8ms), (111.6ms, 113.4ms, 116.6ms), (146.4ms, 148.1ms, 151.1ms) and (146.6ms, 148.3ms, 151.7ms). Furthermore, (106.2ms, 122.5ms), (141.2ms, 157.3ms), (142.2ms, 157.3ms), (106.3ms, 124.2ms), (141.6ms, 158.2ms) and (141.5ms, 159.3ms), were the whiskers value (whisklo, whiskhi) from left to right. In the centralized_hc, we measured higher latency values. For the quartiles (q1, q2, q3) the following values were measured (from left to right): (161.2ms, 177.2ms, 339.3ms), (217.8ms, 270.6ms, 382.8ms), (217.3ms, 252.4ms, 369.7ms), (161.4ms, 176.3ms, 331.4ms), (217.2ms, 253.8ms, 379.1ms) and (217.2ms, 249.7ms, 362.7ms). The whiskers (whisklo, whiskhi) values were (from left to right): (156.0ms, 606.3ms), (209.1ms, 626.9ms), (209.1ms, 597.0ms), (155.6ms, 586.1ms), (208.3ms, 621.4ms) and (208.8ms, 580.3ms).

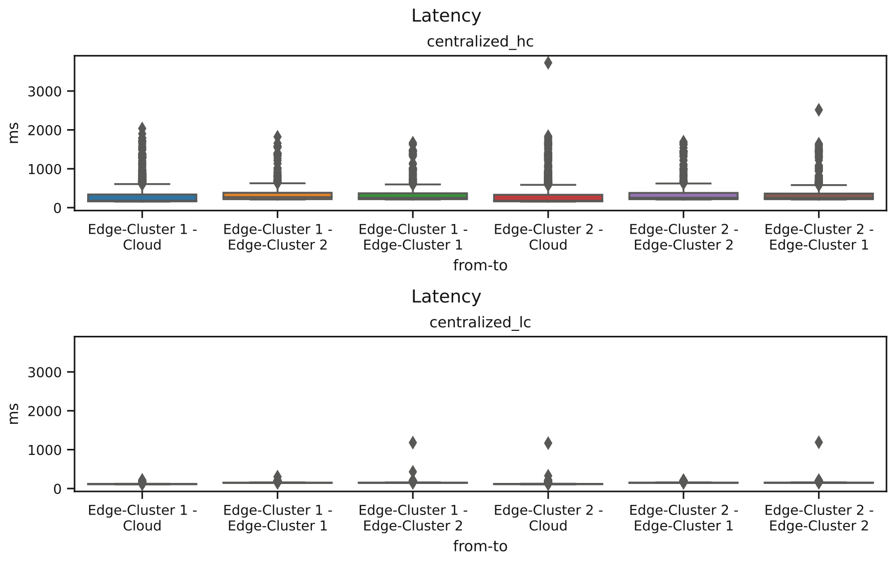


Figure 5.68: Distribution of latency values for each request path in centralized architecture, low and high constraint environment

In Figure 5.69, we see that the mean network throughput of the autoscaler in both centralized_hc and centralized_lc were nearly identical with a mean value of 0.2 MB/s and a standard deviation of 0.03 MB/s.

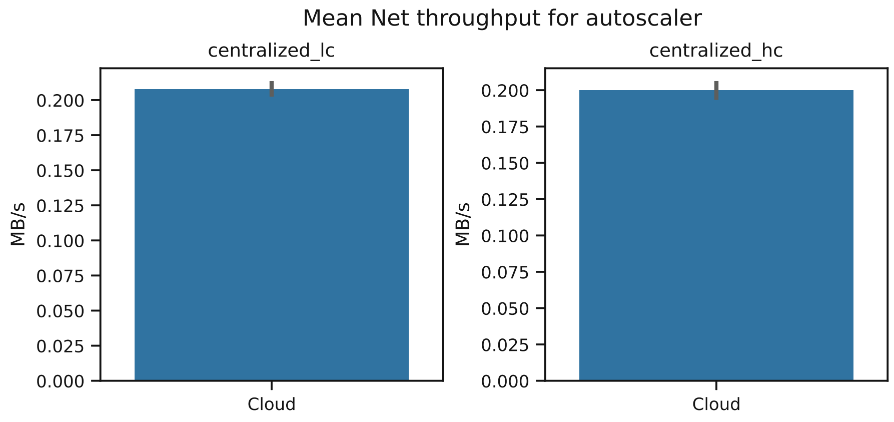


Figure 5.69: Average Net throughput of autoscaler in centralized architecture, low and high constraint environment

Figure 5.70 describes the mean net throughput of function replicas in both infrastructures with a centralized architecture. In centralized_lc, the highest mean value was 0.66 MB/s with a standard deviation of 0.65 MB/s in the Cloud. Edge-Cluster 2 had the lowest with 0.26 MB/s and a standard deviation of 0.3 MB/s. Similarly to centralized_lc, centralized_hc had the highest measured mean value in the Cloud with 0.42 MB/s and a standard deviation of 0.45 MB/s. However, the lowest mean value was in Edge-Cluster 1, with 0.31 MB/s and a standard deviation of 0.34 MB/s.

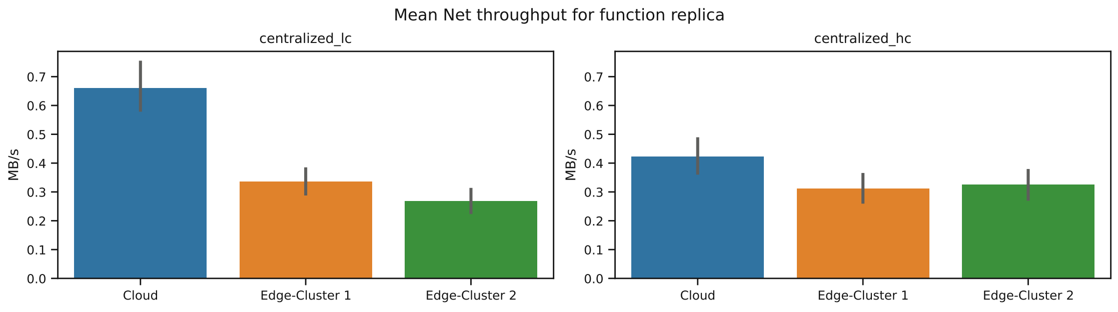


Figure 5.70: Average Net throughput of function replica in centralized architecture, low and high constraint environment

The following figure, Figure 5.71, represents the average network throughput of load-balancer in centralized_lc and centralized_hc. Between these two architectures, there is a recognizable difference, where in centralized_lc, the mean throughput was at 28.5 MB/s with a standard deviation of 22.6 MB/s, and in centralized_hc, 21.8 MB/s with a standard deviation of 15.9 MB/S.

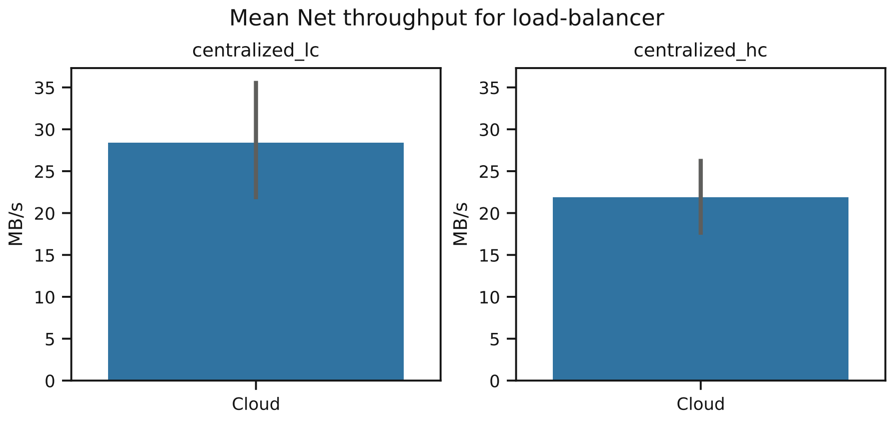


Figure 5.71: Average Net throughput of load-balancer in centralized architecture, low and high constraint environment

Figure 5.72 illustrates the average network throughput of a scheduler for a centralized architecture in both a low and high constraint environment. The mean value in centralized_lc was 0.24 MB/s with a standard deviation of 0.03 MB/s. This value was higher than in centralized_hc, where it was 0.2 MB/s, with a standard deviation of 0.05 MB/s.

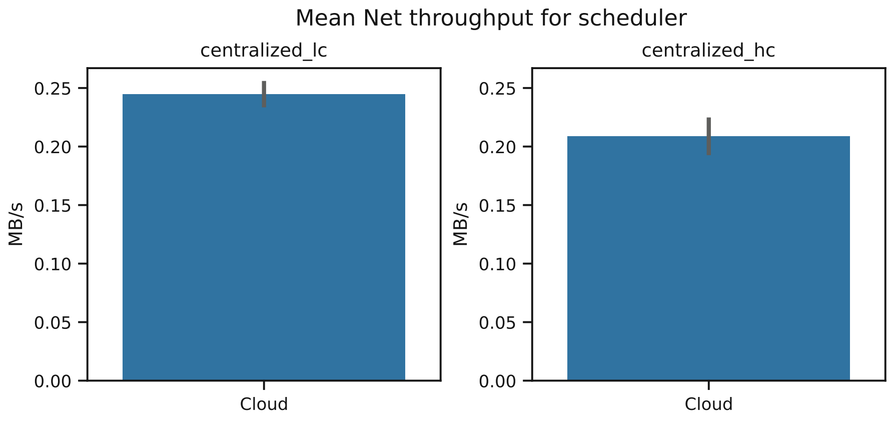


Figure 5.72: Average Net throughput of scheduler, low constraint in centralized architecture, low and high constraint environment

5.3.9 Network level *hc_lc* parameter set with decentralized architecture

Figure 5.73 represents the network throughput distribution of the decentralized architecture in both high and low environments. We see that decentralized_lc had (0.1MB/s, 0.1MB/s, 2.8MB/s) as quartiles (q1,q2,q3), and (0.0MB/s, 7.0MB/s) as whiskers (whisklo, whiskhi). Very similar values were measured in distributed_hc with the following quartiles values: (0.0MB/s, 0.1MB/s, 3.1MB/s) and whiskers: (0.0MB/s, 7.8MB/s).

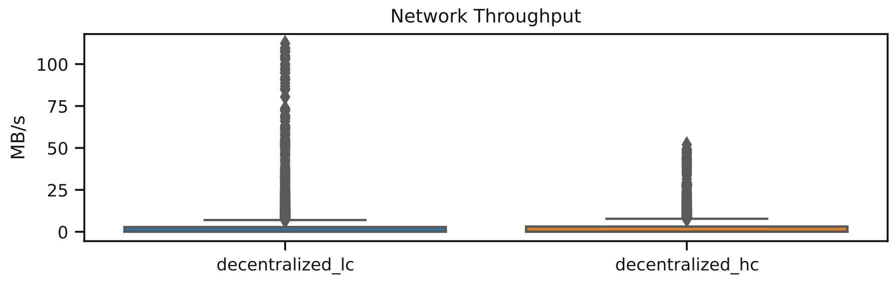


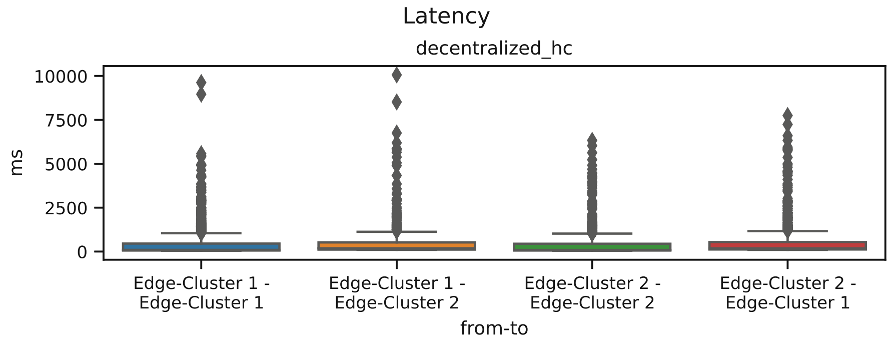
Figure 5.73: Distribution Network Throughput per second per infrastructure, high and low constraint environment, decentralized architecture

In Figure 5.74, we illustrate the distribution of the latency of user requests in decentralized architecture for both low and high constraint environments as box plots. The user requests are further categorized after their four request path. From left to right, for decentralized_lc the quartiles (q1, q2, q3) were as follows: (41.7ms, 44.2ms, 54.6ms), (78.1ms, 107.1ms, 167.3ms), (41.9ms, 44.6ms, 55.1ms) and (82.7ms, 107.3ms, 160.3ms). Regarding the whiskers (whisklo, whiskhi), the values were as follows (from left to right): (36.2ms, 73.9ms), (68.7ms, 300.8ms), (36.0ms, 74.9ms) and (68.5ms, 275.6ms). The quartiles and whiskers for the box plots in decentralized_hc were higher than this. From left to right, the quartiles were: (59.1ms, 95.1ms, 454.2ms), (114.4ms, 202.7ms, 513.9ms), (58.9ms, 89.2ms, 450.1ms) and (117.5ms, 178.3ms, 538.8ms), and the whiskers were: (51.1ms, 1044.0ms), (99.0ms, 1072.3ms), (51.2ms, 1036.6ms), (98.5ms, 1164.8ms).

Looking at Figure 5.75, we see that the average network throughput for the autoscalers in decentralized_lc and decentralized_lc were both relatively close to each other. In decentralized_lc, the autoscalers in each cluster had a mean throughput of around 16 MB/s with a standard deviation of around 0.03 MB/s. In decentralized_hc, the mean network throughput was higher, with around 0.17 MB/s and a standard deviation of around 0.02 MB/s for each cluster.

Figure 5.76 illustrates the average network throughput of a function replica in both environments with a decentralized architecture. We can see a rather significant difference between the two infrastructures. In decentralized_lc, we measured a mean value of 4.1 MB/s with a standard deviation of 4.3 MB/s in Edge-Cluster 1 and 3.5 MB/s with a standard deviation of 4.1 MB/s in Edge-Cluster 2. On the other hand, in decentralized_hc, we only have 0.95 MB/s with a standard deviation of 0.7 MB/s in Edge-Cluster 1 and 1.3 MB/s with a standard deviation of 0.9 MB/s in Edge-Cluster 2.

In Figure 5.77, we represent the average network throughput of load-balancers in decentralized_hc and decentralized_lc. In decentralized_lc, the lowest mean value was measured in the Cloud with 2.3 MB/s and a standard deviation of 0.5 MB/s. The other two clusters had a very similar mean usage, with a value of around 25.5 MB/s and a standard deviation of around 24 MB/s. The same pattern can be observed in decentralized_hc. The Cloud also had the lowest mean network throughput of 0.2 MB/s



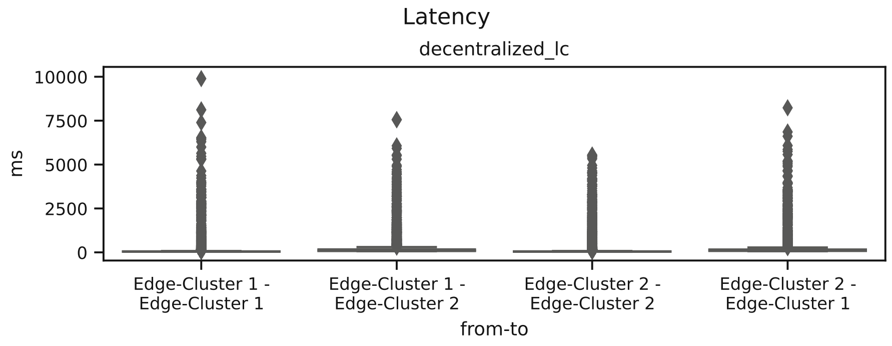


Figure 5.74: Distribution of latency values for each request path in decentralized architecture, low and high constraint environment

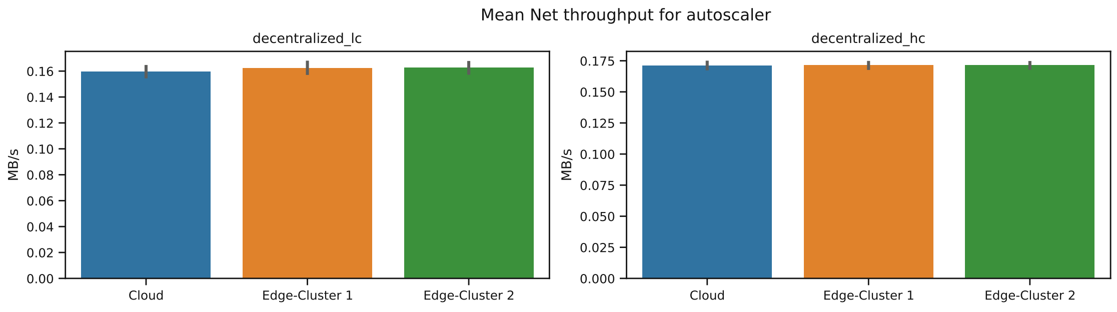


Figure 5.75: Average Net throughput of autoscaler in decentralized architecture, low and high constraint environment

with a standard deviation of 0.27 MB/s. Edge-Cluster 1 and 2 also had a similar mean throughput of around 19 MB/s with a standard deviation of around 14 MB/s.

Figure 5.78 describes the average network throughput of the schedulers in each cluster of the decentralized_lc and decentralized_hc infrastructure. We can see that all three clusters in decentralized_lc had a nearly identical mean throughput of around 0.2 MB/s with a standard deviation of around 0.05 MB/s. Similarly, in decentralized_hc, each cluster had a similar mean network throughout. The mean network throughput was around 0.18 MB/s with a standard deviation of 0.02 MB/s.

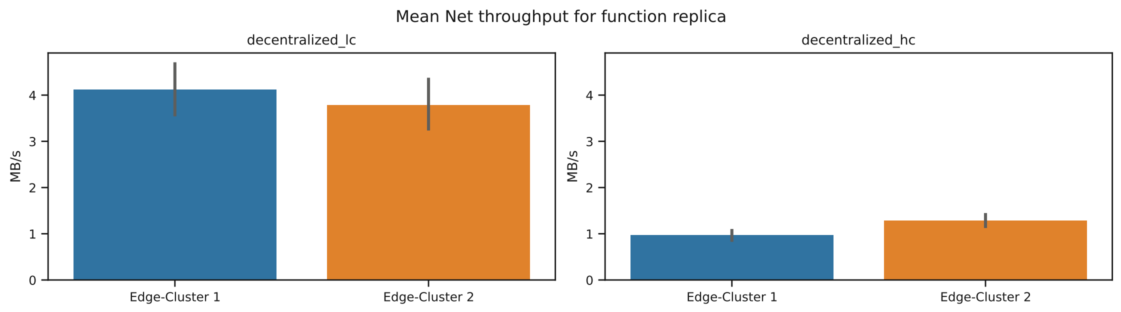


Figure 5.76: Average Net throughput of function replica in decentralized architecture, low and high constraint environment

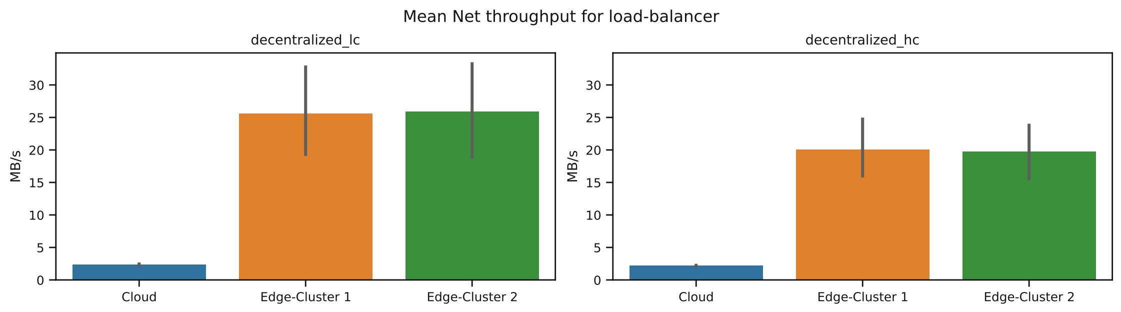


Figure 5.77: Average Net throughput of load-balancer in decentralized architecture, low and high constraint environment

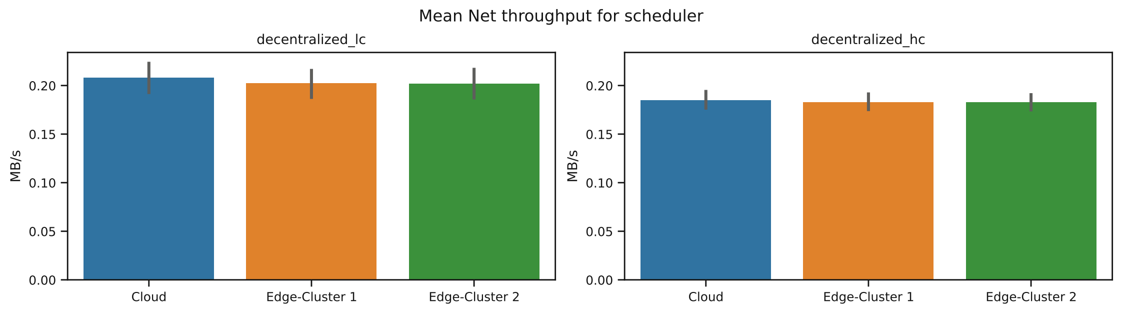


Figure 5.78: Average Net throughput of scheduler, low constraint in decentralized architecture, low and high constraint environment

5.3.10 Network level *hc_lc* parameter set with distributed architecture

The following Figure 5.79 illustrates the network throughput per second distribution of both environments with distributed architecture. For distributed_lc the quartiles (q1, q2, q3) were (0.0MB/s, 0.1MB/s, 3.8MB/s) and the box plots' whiskers (whisklo, whiskhi) were (0.0MB/s, 9.3MB/s). Somewhat similar was distributed_hc with (0.1MB/s, 0.4MB/s, 4.7MB/s) as its quartile values and (0.0MB/s, 11.6MB/s) as its low and high whiskers.

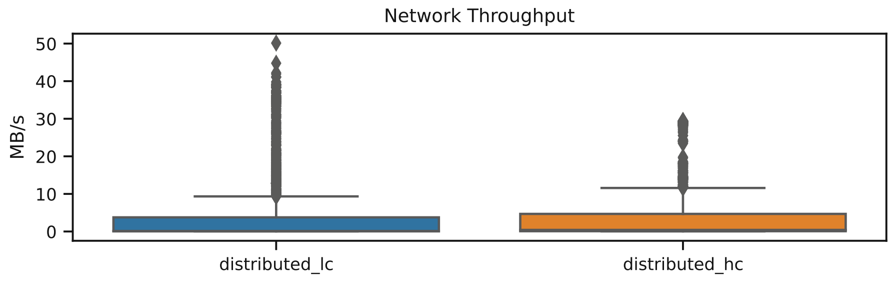


Figure 5.79: Distribution Network Throughput per second per infrastructure, high and low constraint environment, distributed architecture

Figure 5.80 represents the distribution of the latency values in both environments with distributed architecture. Each box plot describes a user request path. From left to right, the following quartile values (q1, q2, q3) were measured in distributed_lc: (41.6ms, 43.3ms, 46.5ms), (41.4ms, 43.0ms, 46.9ms) and (41.7ms, 43.5ms, 47.6ms). For the box plot whiskers (whisklo, whiskhi), the following values were found: (35.8ms, 53.8ms), (35.9ms, 55.2ms) and (36.3ms, 56.4ms). For distributed_hc, the values for both quartiles and whiskers were higher. The following quartiles: (57.8ms, 74.5ms, 456.1ms), (57.6ms, 74.3ms, 465.7ms) and (57.3ms, 73.9ms, 466.8ms), and whiskers:(51.4ms, 1052.3ms), (51.2ms, 1077.2ms) and (51.3ms, 1057.6ms) were calculated for distributed_hc.

Looking at the bar graphs in Figure 5.81, we see that in distributed_lc, the mean network throughput of the autoscaler in each cluster was nearly identical. The mean value for each cluster was around 0.17 MB/s with a standard deviation of around 0.5 MB/s. We have a similar case in distributed_hc, where each autoscaler in the clusters has a similar mean network throughput. The value was around 0.26 MB/s with a standard deviation of around 0.03 MB/s.

In Figure 5.82, we see a function replica's mean network throughput in low and high constraint environments using the distributed architecture. In distributed_lc, the highest mean value was in Edge-Cluster 3 with 4 MB/s and a standard deviation of 3.2 MB/s. The lowest mean value was in Edge-Cluster 2, with 3.1 MB/s and a standard deviation of 2.9 MB/s. In distributed_hc, the highest mean network throughput was in Edge-Cluster 2 with 0.9 MB/s and a standard deviation of 0.9 MB/s. The lowest mean was found in the Edge-Cluster 3 with 0.69 MB/s and a standard deviation of 2.8 MB/s.

In Figure 5.83, we see the mean network throughput of the load-balancers in each cluster in distributed_lc and distributed_hc. The highest mean value in distributed_lc was found in Edge-Cluster 2 with 16.8 MB/s and a standard deviation of 13.3 MB/s, and the lowest in Edge-Cluster 3 with 15.6 MB/s and a standard deviation of 12.5 MB/s. The same pattern can be found in distributed_hc. The cluster with the highest mean network throughput was Edge-Cluster 2, with 18 MB/s and a standard deviation of 11.4 MB/s.

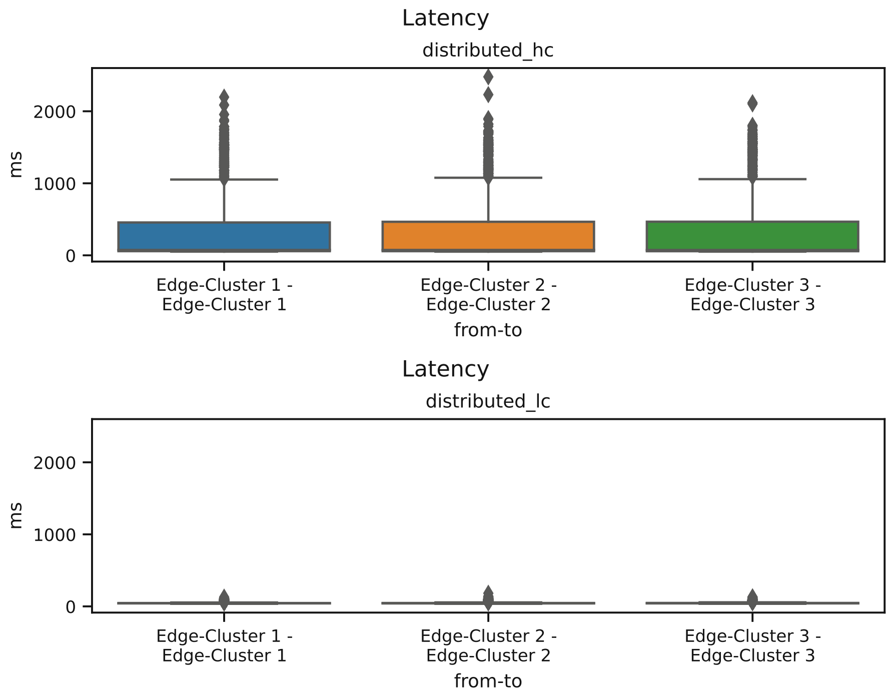


Figure 5.80: Distribution of latency values for each request path in a distributed architecture, low and high constraint environment

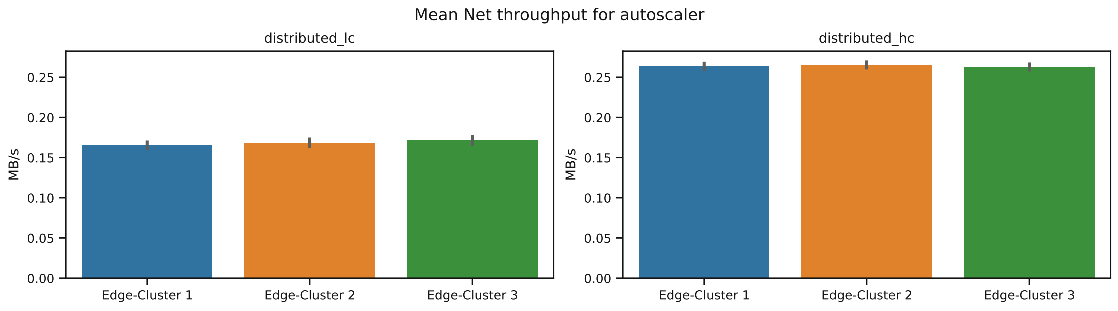


Figure 5.81: Average Net throughput of autoscaler in a distributed architecture, low and high constraint environment

Edge-Cluster 3 had the lowest mean value, with 16.6 MB/s and a standard deviation of 10.7 MB/s.

Figure 5.84 illustrates the average network throughput of schedulers in each cluster of distributed_lc and distributed_hc infrastructure. In distributed_lc, schedulers in all clusters had a similar mean network throughput of around 0.19 MB/s with a standard deviation of around 0.07 MB/s. In contrast, the mean values are more different for the distributed_hc infrastructure. There, a scheduler's highest mean network throughput was in Edge-Cluster 2 with 0.32 MB/s and a standard deviation of 0.05 MB/s. The

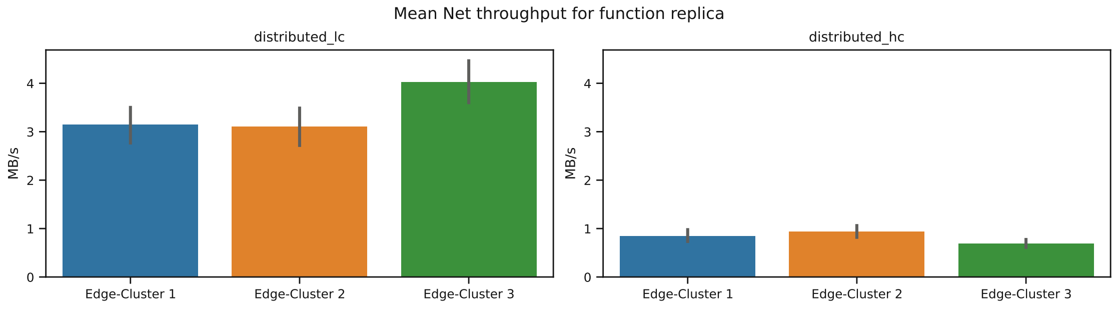


Figure 5.82: Average Net throughput of function replica in a distributed architecture, low and high constraint environment

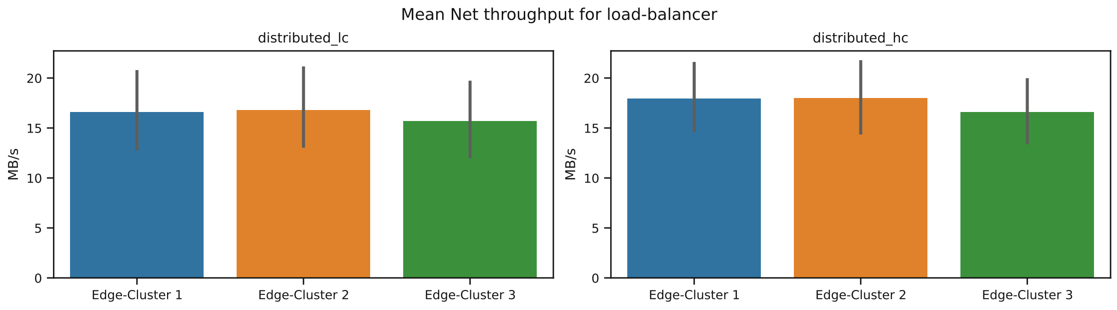


Figure 5.83: Average Net throughput of load-balancer in a distributed architecture, low and high constraint environment

lowest average value was in Edge-Cluster 3, with 0.3 MB/s and a standard deviation of 0.05 MB/s.

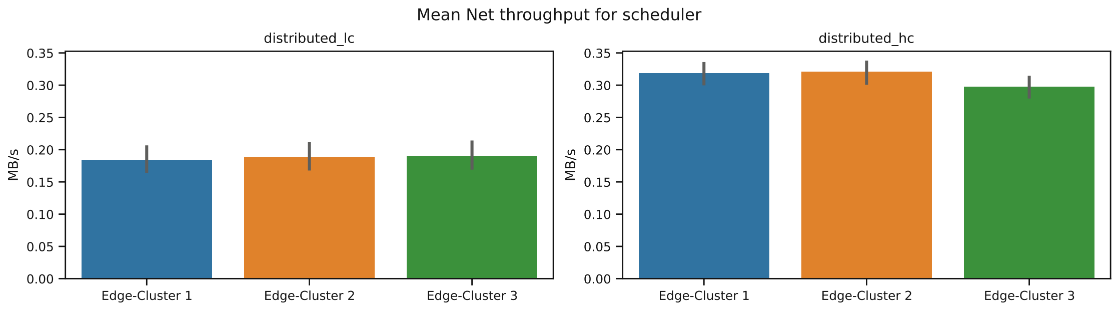


Figure 5.84: Average Net throughput of scheduler, low constraint in distributed architecture, low and high constraint environment

5.3.11 Application level *hc* parameter set

Figure 5.85 illustrates the round-trip-time (rtt) values distribution for all three architectures in high constraint infrastructures as box plots. Each box plot represents a user request path. In decentralized_hc, four different request paths were found, and the following quartiles (q1,q2,q3) were calculated (from left to right): (0.3s, 0.5s, 3.0s), (0.6s, 0.9s, 4.7s), (0.3s, 0.5s, 3.1s) and (0.5s, 0.7s, 5.0s). The box plot whiskers (whisklo, whiskhi) for this infrastructure were as follows (from left to right): (0.3s, 6.9s), (0.4s, 10.8s), (0.3s, 7.2s) and (0.4s, 11.6s). In distributed_hc, the quartiles and whiskers were nearly identical across the three request paths. For the quartiles, the following values were measured (from left to right): (0.3s, 0.5s, 2.8s), (0.3s, 0.5s, 2.9s) and (0.3s, 0.5s, 2.7s), and for the whiskers (0.3s, 6.4s), (0.3s, 6.8s) and (0.3s, 6.4s) were measured (from left to right). Similarly, in centralized_hc, the quartiles and whiskers were similar across six request paths. For the quartiles, the following values were measured (from left to right): (0.7s, 1.1s, 3.0s), (0.9s, 1.3s, 3.5s), (0.8s, 1.2s, 3.3s), (0.7s, 1.1s, 2.9s), (0.9s, 1.4s, 3.6s) and (0.8s, 1.2s, 3.3s). The box plot whiskers (whisklo, whiskhi) for this infrastructure were as follows (from left to right): (0.7s, 6.4s), (0.7s, 7.3s), (0.7s, 6.9s), (0.7s, 6.2s), (0.7s, 7.7s) and (0.7s, 6.9s).

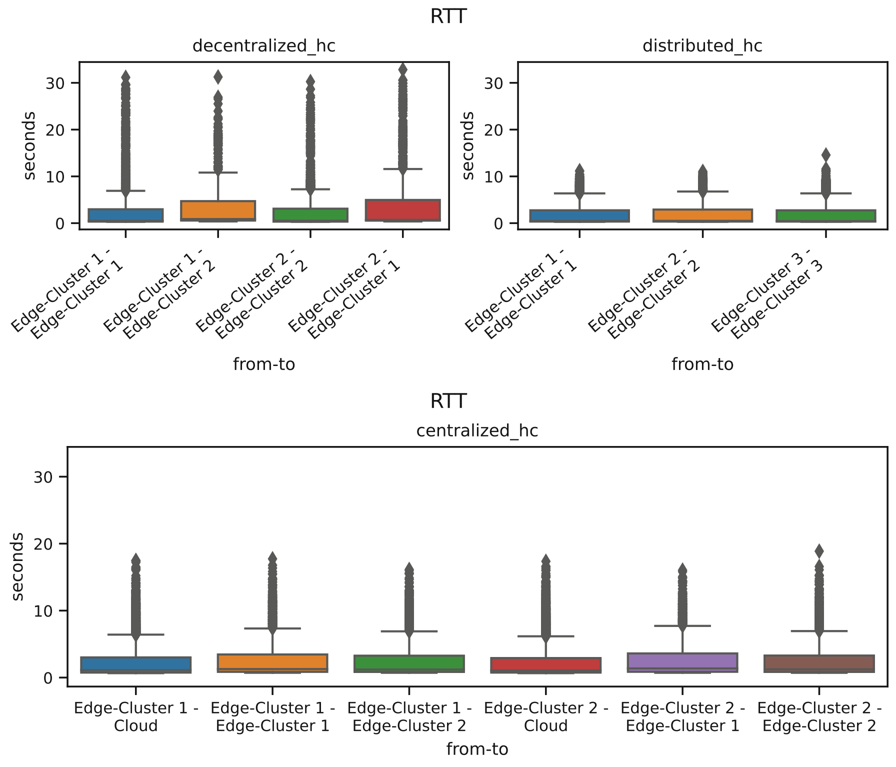


Figure 5.85: Distribution of round-trip-time values for each request path per architecture, high constraint infrastructure, all architectures

We illustrate two different plots in Figure 5.86. On the left side is the actual client request progress throughout the experiment. We can see that the spikes in the line plot are similar to the spikes depicted in Figure 5.4. On the right side, we represent the average number of processed requests per second for all three architectures in a high constraint environment. The infrastructure with the lowest requests per second was centralized_hc, with a mean value of 12.3 and a standard deviation of 7.1. The highest one was distributed_hc with a mean value of 19.4 and a standard deviation of 12.5.

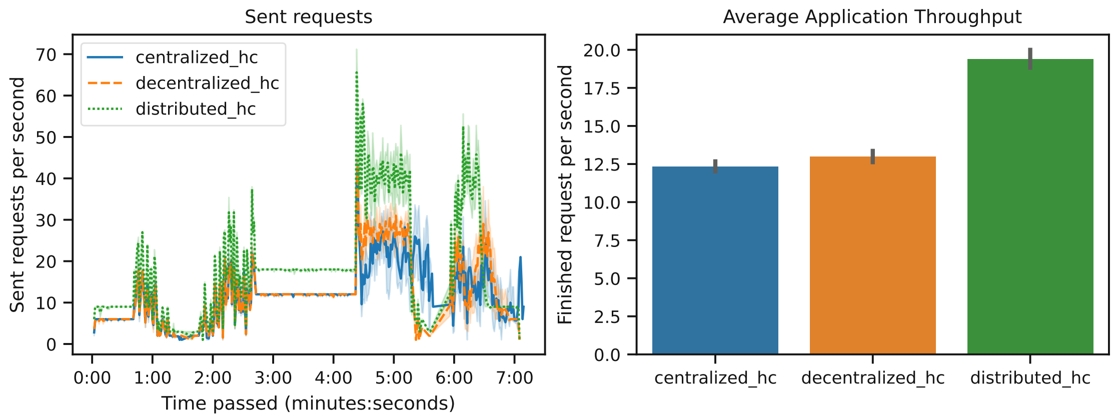


Figure 5.86: Sent request progress and average finished requests per architecture, high constraint infrastructure, all architectures

5.3.12 Application level *lc* parameter set

In order to illustrate the rtt values, we created box plots. We created one for each request path in each experiment, which can be seen in Figure 5.87. In decentralized_lc, the following quartiles (q1, q2, q3) were found (from left to right): (0.3s, 0.5s, 2.2s), (0.6s, 2.7s, 6.2s), (0.3s, 0.5s, 2.7s) and (0.9s, 2.6s, 6.2s). The box plot whiskers (whisklo, whiskhi) were as follows (from left to right): (0.2s, 5.2s), (0.3s, 14.6s), (0.2s, 6.4s) and (0.3s, 14.0s). For distributed_lc, both the quartiles and whiskers were lower. The quartiles were (from left to right): (0.2s, 0.3s, 0.7s), (0.2s, 0.3s, 0.6s) and (0.2s, 0.3s, 1.0s), and whiskers were (from left to right): (0.2s, 1.4s), (0.2s, 1.1s) and (0.2s, 2.0s). Even smaller boxes were measured in centralized_lc. There, the following quartiles: (0.5s, 0.5s, 0.5s), (0.5s, 0.6s, 0.6s), (0.5s, 0.6s, 0.6s), (0.5s, 0.5s, 0.5s), (0.5s, 0.6s, 0.6s) and (0.5s, 0.6s, 0.6s), and the following whiskers: (0.5s, 0.6s), (0.5s, 0.7s), (0.5s, 0.7s), (0.5s, 0.6s), (0.5s, 0.7s) and (0.5s, 0.7s) were measured.

In order to present the average application throughput, we also added the progress of the sent user requests in Figure 5.88. The spikes and flats match our user profile in the line plot on the right side. Distributed_lc had the highest average application throughput, with 19.5 requests per second and a standard deviation of 14.7. The lowest mean figure was measured in centralized_lc with 11.5 requests per second and a standard deviation of 9.3.

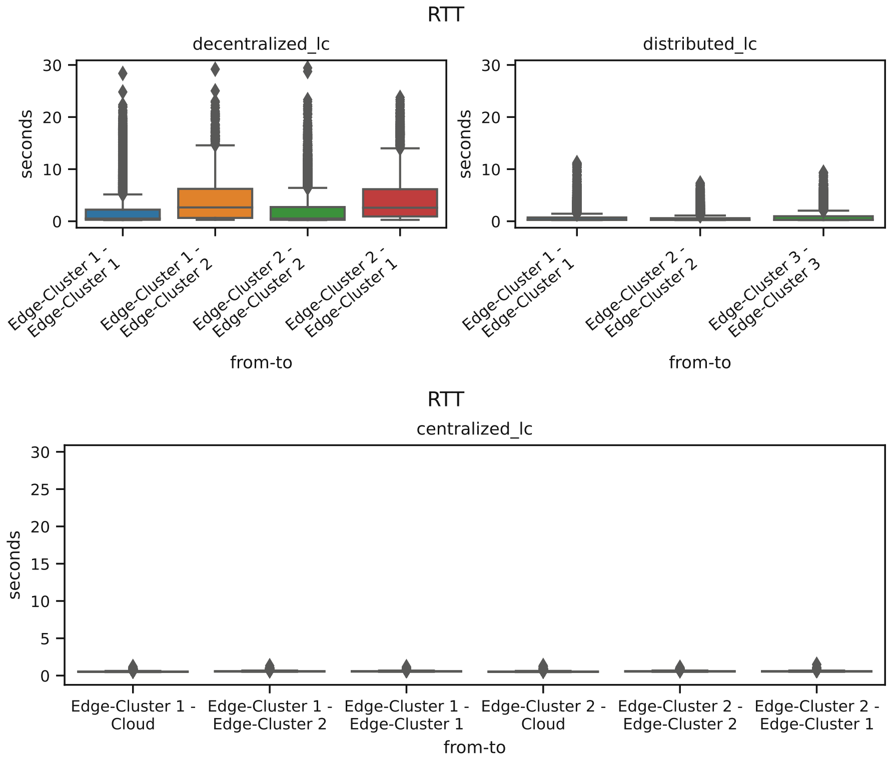


Figure 5.87: Distribution of round-trip-time values for each request path per architecture, low constraint infrastructure, all architectures

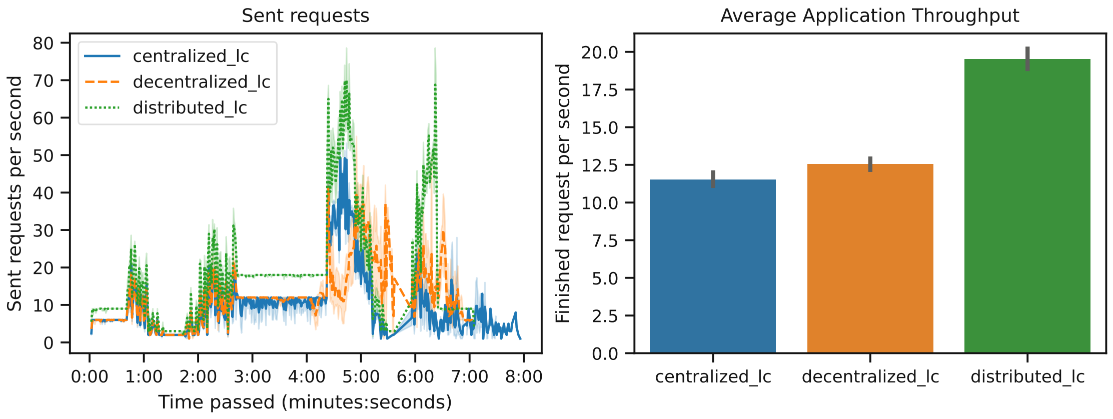


Figure 5.88: Sent request progress and average finished requests per architecture, low constraint infrastructure, all architectures

5.3.13 Application level *hc_lc* parameter set with centralized architecture

Figure 5.89 illustrates the distribution of rtt values across user request paths in centralized_lc and centralized_hc. We can see a difference between the box plot sizes. In centralized_lc, the following quartiles (from left to right) were measured: (0.5s, 0.5s, 0.5s), (0.5s, 0.6s, 0.6s), (0.5s, 0.6s, 0.6s), (0.5s, 0.5s, 0.5s), (0.5s, 0.6s, 0.6s) and (0.5s, 0.6s, 0.6s). For the box plot whiskers (whisklo, whiskhi), in centralized_lc, the following values were found (from left to right): (0.5s, 0.6s), (0.5s, 0.7s), (0.5s, 0.7s), (0.5s, 0.6s), (0.5s, 0.7s) and (0.5s, 0.7s). As mentioned, the quartiles in centralized_hc are more spread out. There we had the following quartiles (from left to right): (0.7s, 0.9s, 1.9s), (0.9s, 1.2s, 1.9s), (0.8s, 1.2s, 1.9s), (0.7s, 0.8s, 1.8s), (0.8s, 1.2s, 1.9s) and (0.8s, 1.1s, 1.8s), and the following whiskers (from left to right): (0.7s, 3.6s), (0.7s, 3.4s), (0.7s, 3.3s), (0.7s, 3.4s), (0.7s, 3.5s) and (0.7s, 3.2s).

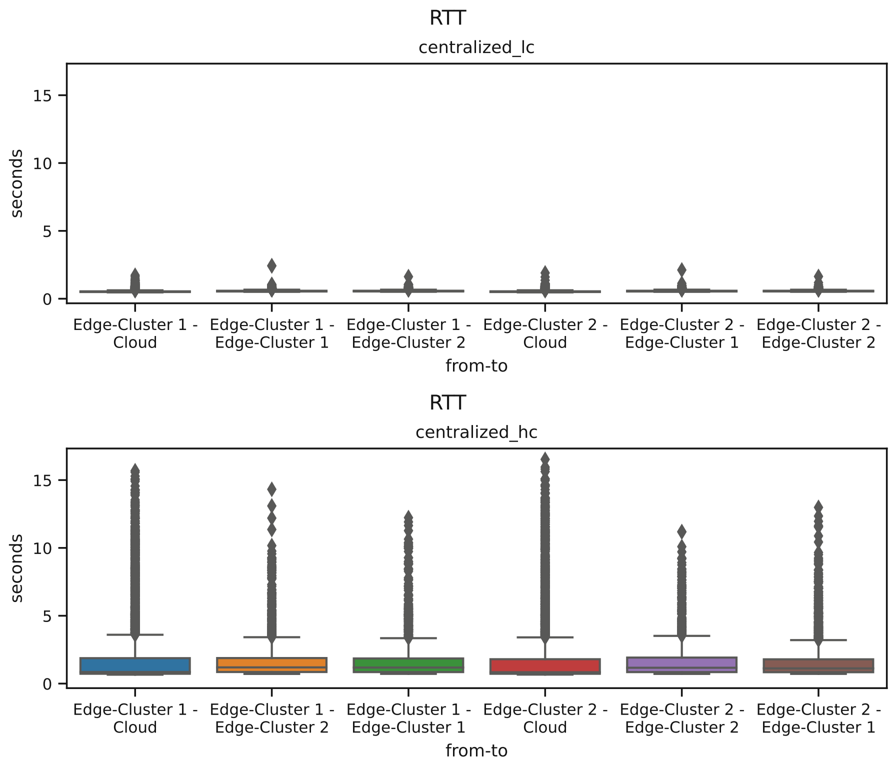


Figure 5.89: Distribution of latency values for each request path in centralized architecture, low and high constraint environment

In Figure 5.90, we illustrate the actual request workload from our clients and the average application throughput per second. The line plot shows that the last spike is less predominant than the user profile suggests. Regarding the average application throughput, we see that it was in centralized_lc, with a mean value of 11.8 requests per second and a standard deviation of 9.2, higher than in centralized_hc, with a mean value of 10.8 requests per second and a standard deviation of 6.7.

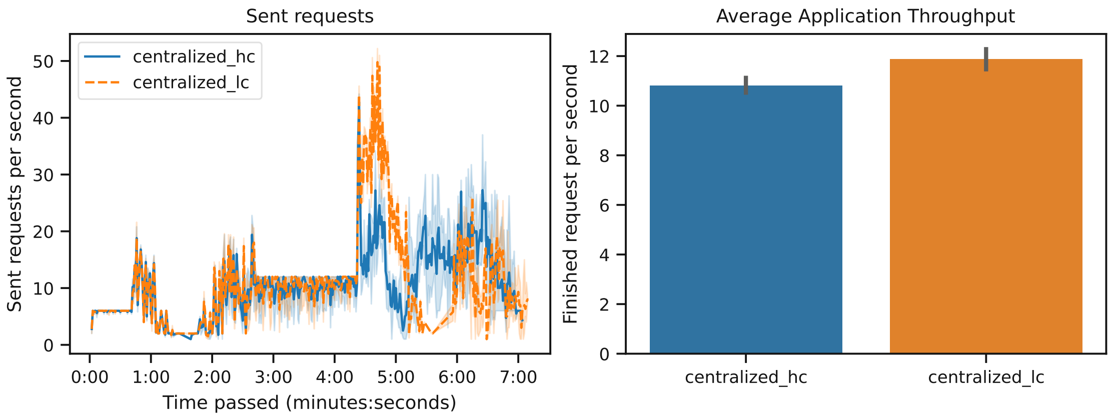


Figure 5.90: Sent request progress and average finished requests in centralized architecture, low and high constraint environment

5.3.14 Application level *hc_lc* parameter set with decentralized architecture

In order to illustrate the distribution of rtt values in both decentralized_lc and decentralized_hc, we created multiple box plots, where each box plot represents a request path (see Figure 5.91). In decentralized_lc, the following quartiles (q1, q2, q3) were measured (from left to right): (0.3s, 0.4s, 1.6s), (0.5s, 1.8s, 5.3s), (0.3s, 0.4s, 1.6s) and (0.8s, 3.0s, 6.6s). The whiskers (whisklo, whiskhi) in the box plots in decentralized_lc were (from left to right): (0.2s, 3.5s), (0.3s, 12.6s), (0.2s, 3.5s) and (0.3s, 15.2s). For decentralized_hc, higher values were measured. The quartiles were at (from left to right): (0.3s, 0.5s, 2.9s), (0.6s, 0.9s, 4.8s), (0.3s, 0.5s, 3.0s) and (0.6s, 0.7s, 5.3s), and the whiskers were at (from left to right): (0.3s, 6.8s), (0.4s, 11.0s), (0.3s, 6.9s) and (0.3s, 12.4s).

Figure 5.92 represents the user request workload and the average application throughput for decentralized_lc and decentralized_hc. While the line plot is similar to our user profile, the difference between the lower and higher spikes was lower than in the user request profile Figure 5.4. Regarding the mean application throughput, both infrastructures had the same mean of 13 requests per second. Decentralized_hc had a standard deviation of 8.1, while decentralized_lc had a standard deviation of 9.5.

5.3.15 Application level *hc_lc* parameter set with distributed architecture

Figure 5.93 illustrates the distribution of rtt values in box plots for distributed_lc and distributed_hc. Each box plot represents a user request path. In distributed_lc, the following quartiles (q1, q2, q3) were found (from left to right): (0.2s, 0.3s, 0.4s), (0.2s,

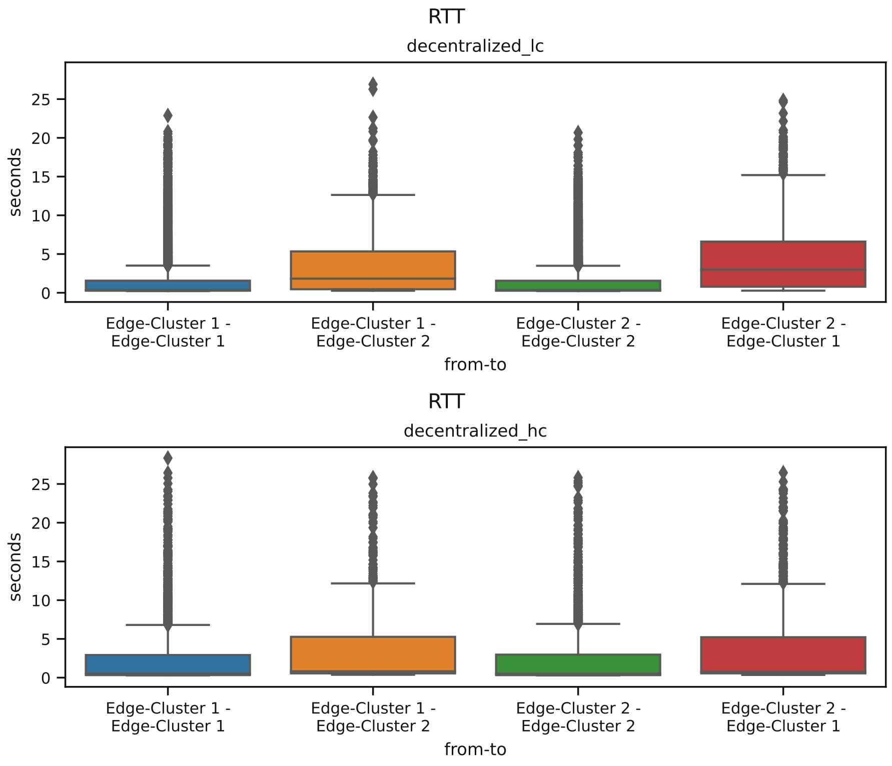


Figure 5.91: Distribution of latency values for each request path in decentralized architecture, low and high constraint environment

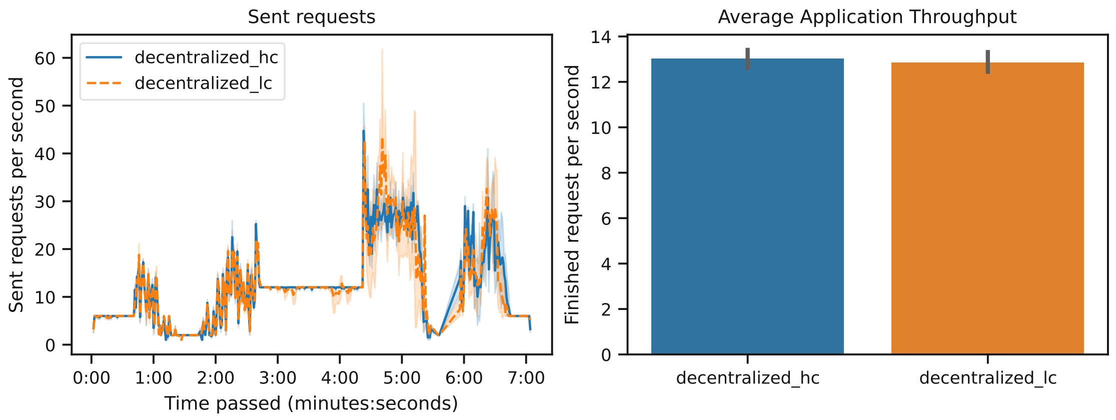


Figure 5.92: Sent request progress and average finished requests in decentralized architecture, low and high constraint environment

0.3s, 0.6s) and (0.2s, 0.3s, 0.7s). Furthermore, these were the whiskers (whisklo, whiskhi) for the box plots in distributed_lc (from left to right): (0.2s, 0.7s), (0.2s, 1.1s), and (0.2s, 1.3s). The quartile and whiskers values in distributed_hc were higher than in distributed_lc. There, the quartiles were (from left to right): (0.3s, 0.5s, 2.9s), (0.3s, 0.5s, 2.9s) and (0.3s, 0.5s, 2.8s), and the whiskers were (from left to right): (0.3s, 6.7s), (0.3s, 6.8s) and (0.3s, 6.4s).

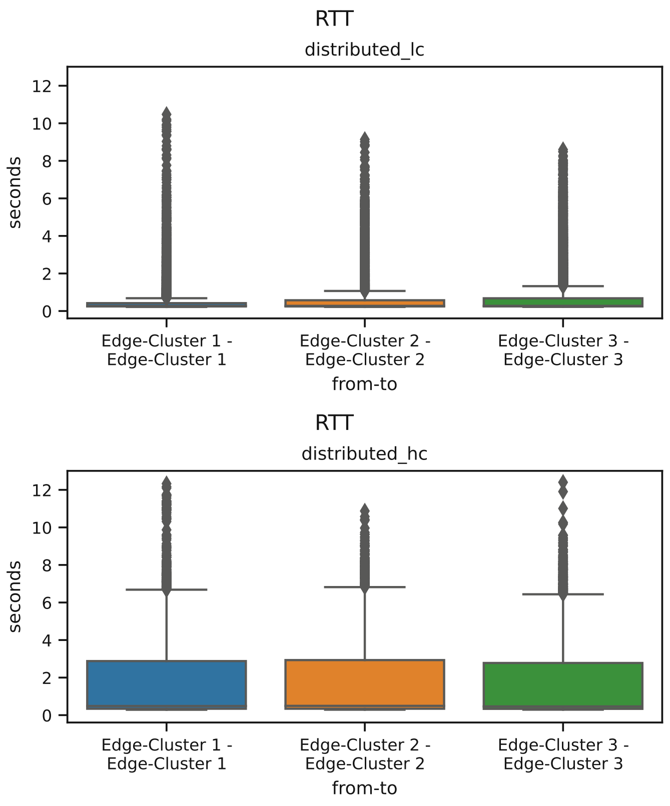


Figure 5.93: Distribution of latency values for each request path in a distributed architecture, low and high constraint environment

The average application throughput and the user request workload are illustrated in Figure 5.94. The request workload is very similar to our created user profiles. For the mean application throughput, both infrastructures had the same mean value of around 19 requests per second. However, the standard deviation is different, where distributed_hc had a value of 12.6, and distributed_lc had a value of 14.7.

5.3.16 Orchestration level *hc* parameter set

Figure 5.95 illustrates each architecture's total zone crossings in high constraint infrastructures. A zone cross happens when a user request's destination and source zone are different. Since there were no zone crossings in distributed_hc, we omitted it from the figure. The bar graph illustrates the mean values of zone crossings over the five experiments. In centralized_hc, the crossing from Edge-Cluster 1 to the Cloud was the most dominant, with a mean occurrence of 1486 and a standard deviation of 80.

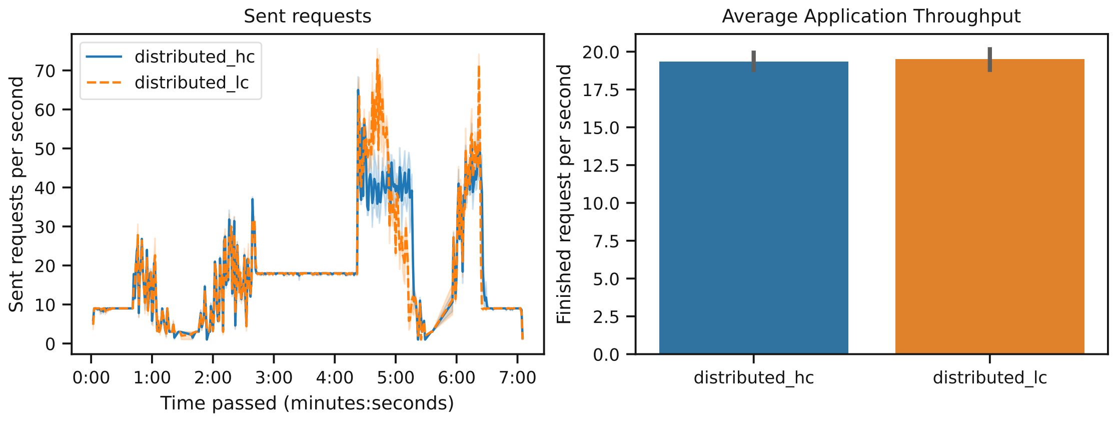


Figure 5.94: Sent request progress and average finished requests in a distributed architecture, low and high constraint environment

The lowest amount was for Edge-Cluster 2 to Edge-Cluster 1, with a mean number of 416 incidents and a standard deviation of 12. In decentralized_hc, the occurrence of Edge-Cluster 2 to Edge-Cluster 1 crossings was, with 257 incidents and a standard deviation of 43, higher than the other way around. The other way happened an average of 184 times with a standard deviation of 36.

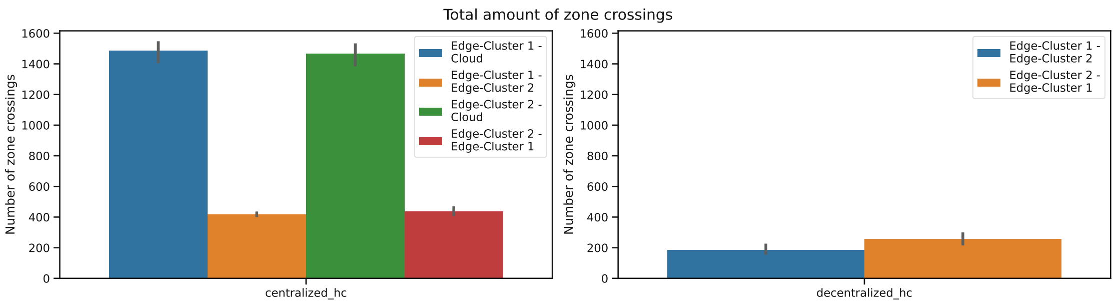


Figure 5.95: Total amount of zone crossings, high constraint infrastructure, all architectures

In Figure 5.96, we illustrate the average scheduling duration of a newly created pod for all three architectures in a high constraint environment. While the lowest duration was in centralized_hc with a mean value of 86.7ms and a standard deviation of 50.7ms, the highest was in decentralized_hc with a mean duration of 255.3ms with a standard deviation of 79.8 ms.

The bar graphs in Figure 5.97 represent the mean scaling decision duration of an autoscaler for all three architectures in a high constraint environment. The autoscalers in distributed_hc took the longest, with an average decision time of 63ms and a standard deviation of 64ms. The quickest decisions were made in decentralized_hc with a mean value of 19.8ms and a standard deviation of 29.8ms.

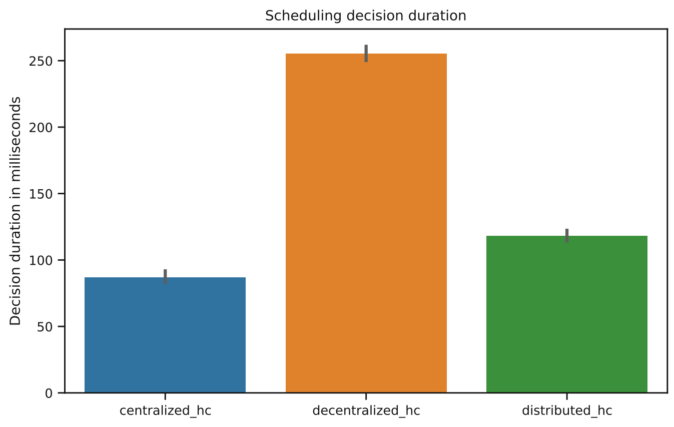


Figure 5.96: Scheduling decision duration for a pod, high constraint infrastructure, all architectures

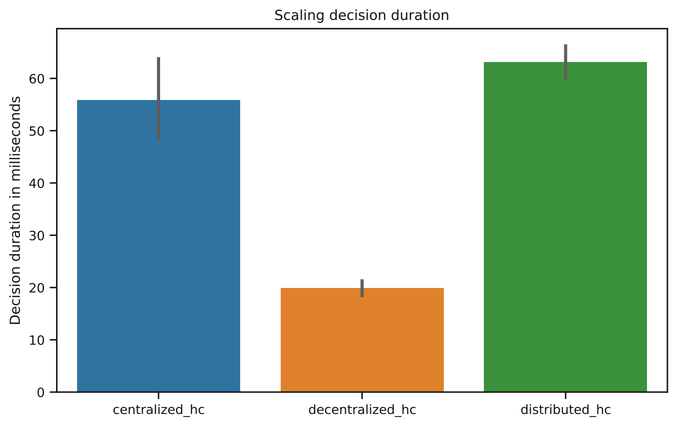


Figure 5.97: Scaling decision duration, high constraint infrastructure, all architectures

Figure 5.98 illustrates the progress of running replicas during the experiments for each architecture with a high constraint infrastructure. In order to create a legible plot, we did a ten-second resample on the number of currently running replicas and took the mean of the samples. The resampling allowed us to create a readable plot that does not distort the actual values too much. All three different setups had a similar scaling pattern. However, centralized_hc scales the highest with up to 90 simultaneously running replicas, while decentralized_hc only reached a maximum of 43 simultaneous replicas and thus had the lowest scaling. Furthermore, in decentralized_hc, no replicas were running in the Cloud cluster.

5.3.17 Orchestration level *lc* parameter set

In Figure 5.99, we illustrate the total zone crossings for our implemented architectures in a low constraint environment. The distributed architecture is omitted since the

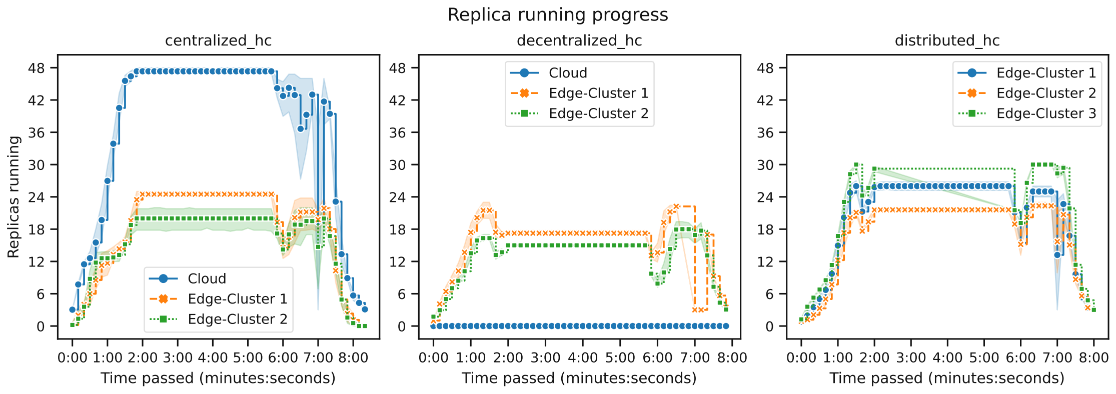


Figure 5.98: Replica running progress, high constraint infrastructure, all architectures

architecture does not support zone crossings. In centralized_lc, the most zone crossings were from Edge-Cluster 1 to the Cloud with a mean number of 1101 times and a standard deviation of 104 times. Zone crossings from Edge-Cluster 1 to Edge-Cluster 2 were the least frequent, with a mean number of 420 times and a standard deviation of 83. In decentralized_hc, both supported zone crossings were relatively close to each other. A zone crossing from Edge-Cluster 1 to Edge-Cluster 2 happened on average 480 times with a standard deviation of 79. In comparison, a crossing from Edge-Cluster 2 to Edge-Cluster 1 happened on average 459 times with a standard deviation of 205.

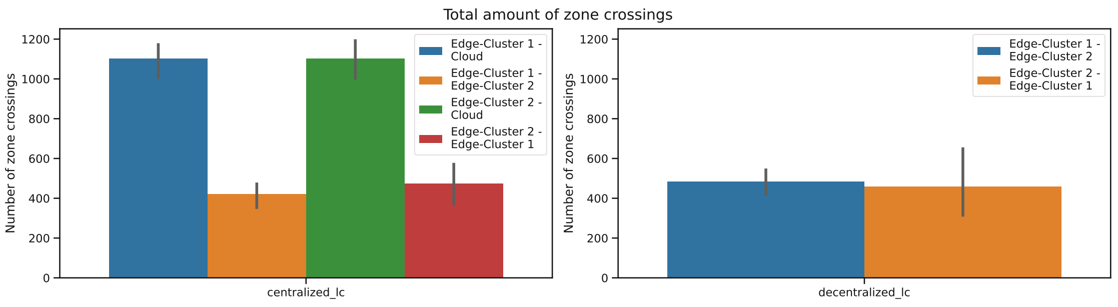


Figure 5.99: Total amount of zone crossings, low constraint infrastructure, all architectures

Figure 5.100 represents how long it took for a pod to complete the scheduling process. The longest scheduling decision was in decentralized_lc with a mean value of 267ms and a standard deviation of 79ms. The shortest duration was in centralized_lc with 89ms and a standard deviation of 23.3ms.

Similarly, Figure 5.101 depicts how long it took for an autoscaler to create a scaling decision. The longest duration was found in centralized_lc with a mean duration of 42.7ms and a standard deviation of 49.2ms. The shortest duration was in decentralized_lc with 24.4ms and a standard deviation of 43.1ms.

Figure 5.102 illustrates the progress of up and down scaling of function replicas for

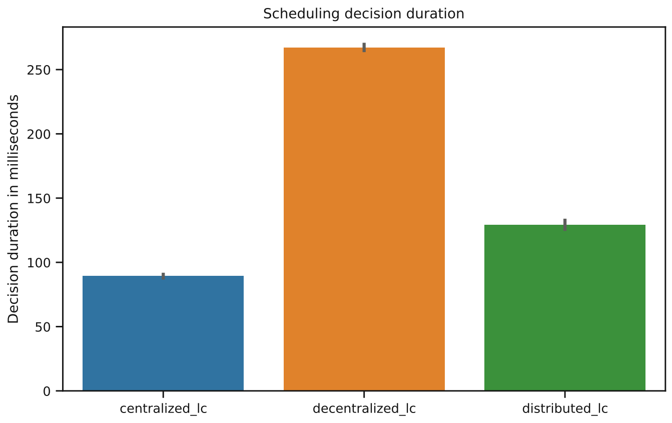


Figure 5.100: Scheduling decision duration for a pod, low constraint infrastructure, all architectures

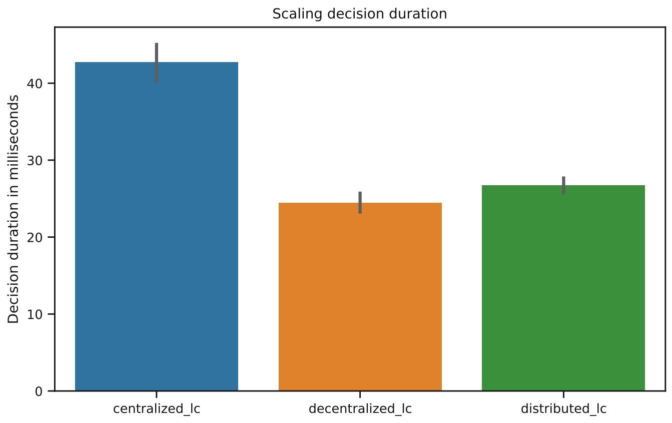


Figure 5.101: Scaling decision duration, low constraint infrastructure, all architectures

each architecture in low constraint infrastructures. The actual data was resampled in a ten-second sample to provide a more legible plot without distorting the data too much. In centralized_hc, the autoscaler scaled aggressively in the first minutes and reached up to 90 concurrent running replicas before going down at the end. Decentralized_lc had a calmer scaling behavior, where a strong upscaling happened at the end of the experiment, reaching a maximum of around 38 concurrent replicas. We could also observe an alternating scaling behavior between Edge-Cluster 1 and Edge-Cluster 2. The scaling trend in distributed_lc was similar to decentralized_lc, with the distinction that the peak in distributed_lc was around 88 simultaneous running replicas higher than in decentralized_lc. Moreover, there was no oscillating scaling pattern in distributed_lc.

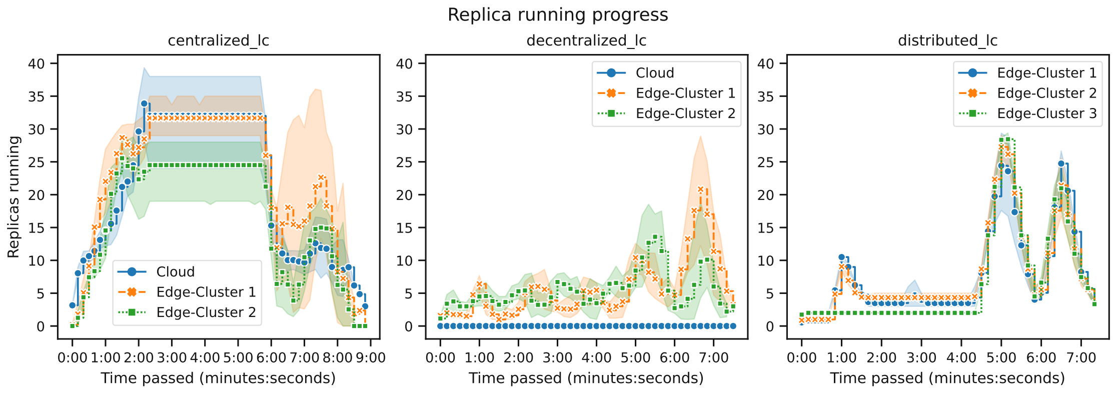


Figure 5.102: Replica running progress, low constraint infrastructure, all architectures

5.3.18 Orchestration level *hc_lc* parameter set with centralized architecture

Figure 5.103 represents the number of zone crossings in centralized architecture for low and high constraint infrastructures. Both plots have a similar pattern. In centralized_lc, the crossing from Edge-Cluster 1 to the Cloud was the highest, with an average amount of 1045 times and a standard deviation of 105 times. The crossing from Edge-Cluster 1 to Edge-Cluster 2 was the lowest occurrence, with an average amount of 699 times and a standard deviation of 51. Centralized_hc has the same pattern, in the sense that the crossing from Edge-Cluster 1 to the Cloud happened the most with around 1199 times and a standard deviation of 78, and Edge-Cluster 1 to Edge-Cluster 2 happened the least amount of times with an average of 386 times and a standard deviation of 106.

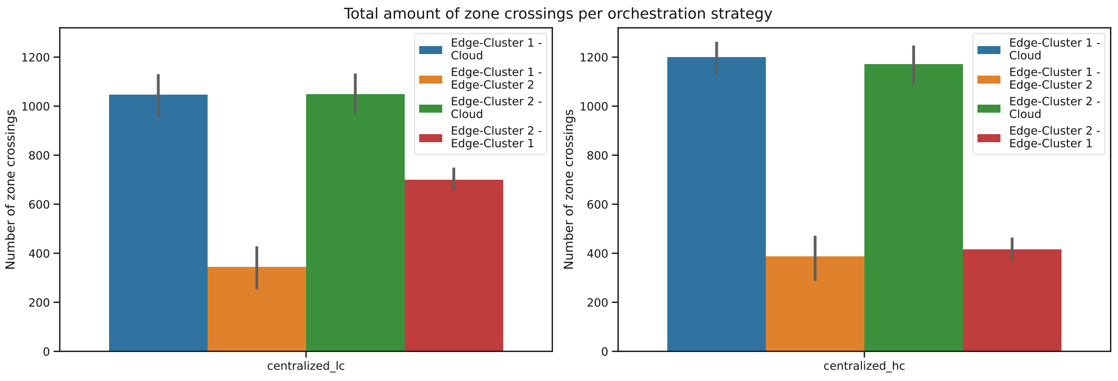


Figure 5.103: Total amount of zone crossings in centralized architecture, low and high constraint infrastructure

The bar graph in Figure 5.104 depicts the average decision duration of the scheduler for scheduling a newly created pod. We see that on average, centralized_lc took, with 88ms and a standard deviation of 24ms, longer for a decision than centralized_hc, with 84ms and a standard deviation of 20.9ms.

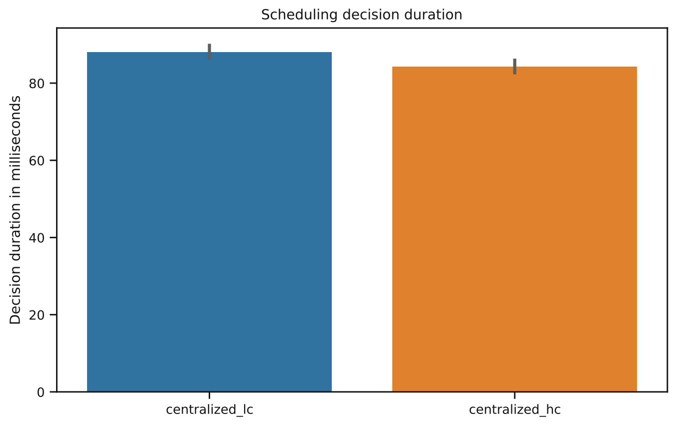


Figure 5.104: Scheduling decision duration for a pod in centralized architecture, low and high constraint infrastructure

Similarly, the bar graphs in Figure 5.105 illustrate the average duration of a scaling decision. While close, centralized_lc took, with 53.3ms and a standard deviation of 97.6ms, longer than centralized_hc, with 23.8 and a standard deviation of 77.9ms.

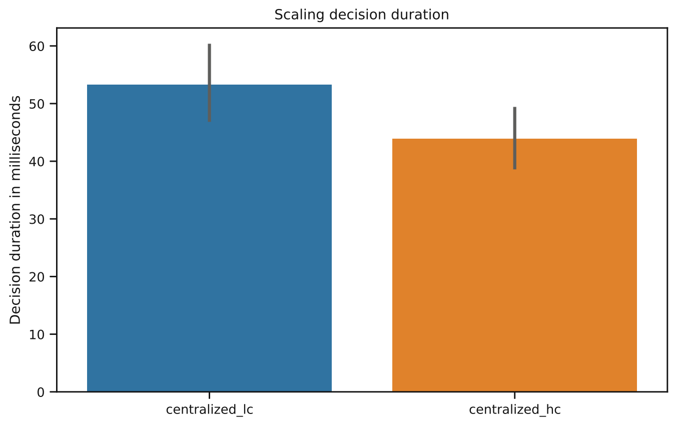


Figure 5.105: Scaling decision duration for a pod in centralized architecture, low and high constraint infrastructure

Figure 5.106 illustrates the running replica development of centralized_hc and centralized_lc. Both had a strong scale-up, with centralized_lc reaching the maximum of 90 concurrent running replicas and centralized_hc reaching 89. Their scale-up, scale-down, and non-scaling phases were nearly identical. While in centralized_hc, the Cloud was the cluster with the most running replicas, in centralized_lc, it was Edge-Cluster 1.

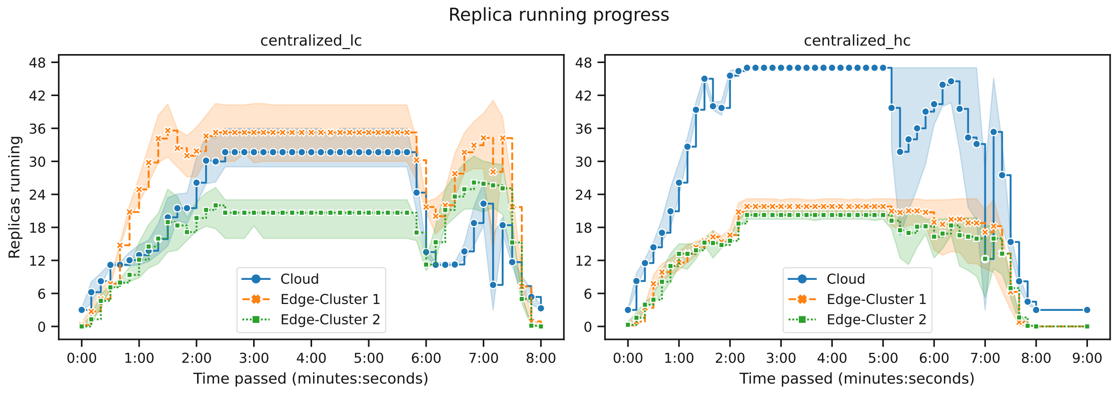


Figure 5.106: Replica running progress in centralized architecture, low and high constraint infrastructure

5.3.19 Orchestration level *hc_lc* parameter set with decentralized architecture

In Figure 5.107, we illustrate the amount of zone crossings user requests did in decentralized architecture in high and low constraint environments. In decentralized_hc and decentralized_lc, no zone crossings to the Cloud cluster took place. In decentralized_lc, the number of times crossings from Edge-Cluster 1 to Edge-Cluster occurred were, on average, 368 times with a standard deviation of 100. The other direction, Edge-Cluster 2 to Edge-Cluster 1, happened on average 314 times, with a standard deviation of 144. In decentralized_hc, crossings from Edge-Cluster 2 to Edge-Cluster 1 happened more often than the other way around. Crossings from Edge-Cluster 2 to Edge-Cluster 1 happened on average 242 times with a standard deviation of 78, and Edge-Cluster 1 to Edge-Cluster crossings occurred on average 199 times with a standard deviation of 47.

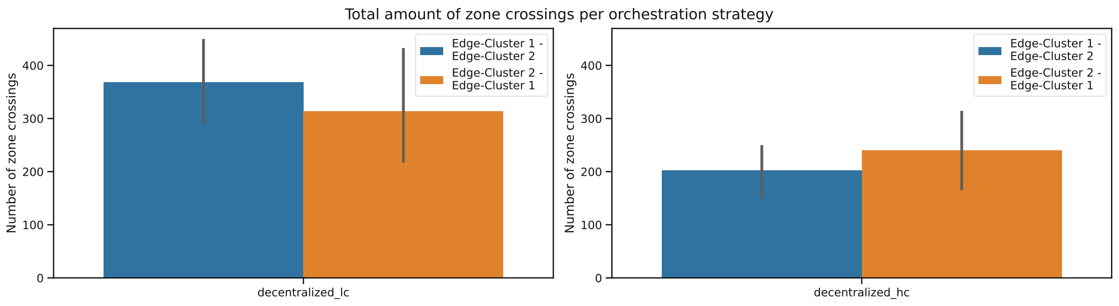


Figure 5.107: Total amount of zone crossings in decentralized architecture, low and high constraint infrastructure

Figure 5.108 represents the average scheduling decision of a pod in decentralized architecture in low and high constraint infrastructure. We see that in both cases, the duration was nearly identical. In decentralized_lc, the scheduling of a pod took on average 262ms with a standard deviation of 130ms; in decentralized_hc, it took on average 258ms with a standard deviation of 78ms.

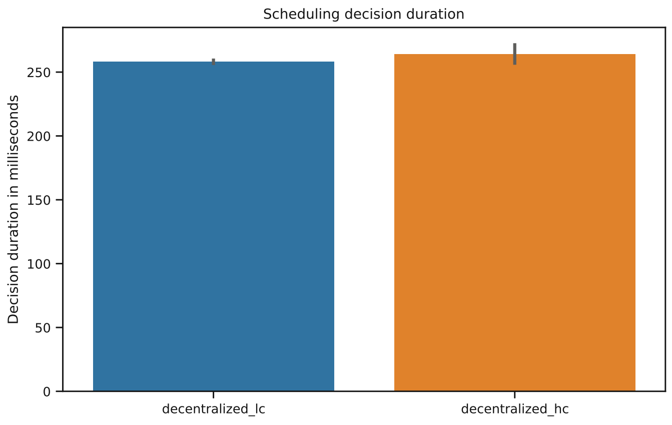


Figure 5.108: Scheduling decision duration for a pod in decentralized architecture, low and high constraint infrastructure

The bar graphs in Figure 5.108 depict the average scaling decision in both infrastructures with a decentralized architecture. We see that a scaling decision in decentralized_hc took, on average, 29.3ms with a standard deviation of 76.1ms, while in decentralized_lc, a scale decision was made in only 18.6ms with a standard deviation of 30.1ms.

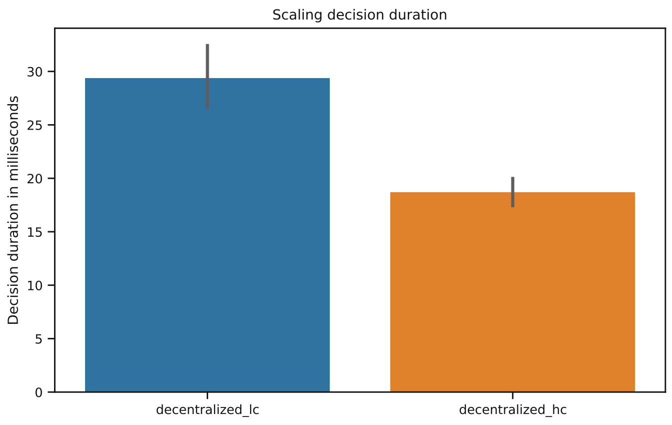


Figure 5.109: Scaling decision duration for a pod in decentralized architecture, low and high constraint infrastructure

With Figure 5.110, we illustrate the progress of running replicas throughout the experiments in a high and low constraint environment with a decentralized architecture. Both had a similar max replica count, with decentralized_lc being at 44 and decentralized_hc being at 40. Furthermore, in decentralized_hc, a significant scale-up happened two times, early and late into the experiments, while in decentralized_lc, a significant scale-up only happened late into the experiment. Additionally, decentralized_hc had more replicas running for a longer time than decentralized_lc.

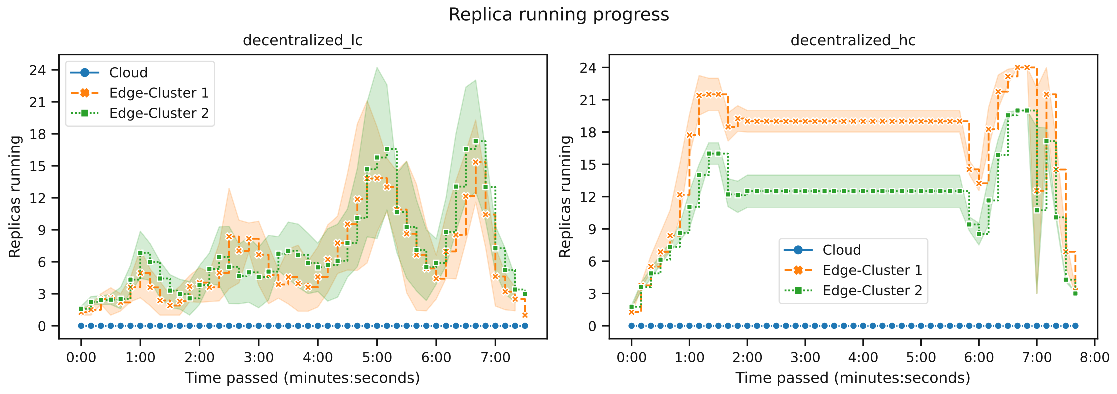


Figure 5.110: Replica running progress in decentralized architecture, low and high constraint infrastructure

5.3.20 Orchestration level *hc_lc* parameter set with distributed architecture

With a distributed architecture, the schedulers in both infrastructures took nearly identical time to schedule a new function replica. Looking at Figure 5.111, we see that the schedulers in distributed_lc took on average 128.4ms with a standard deviation of 53ms, and in distributed_hc 123.8ms with a standard deviation of 53.1ms.

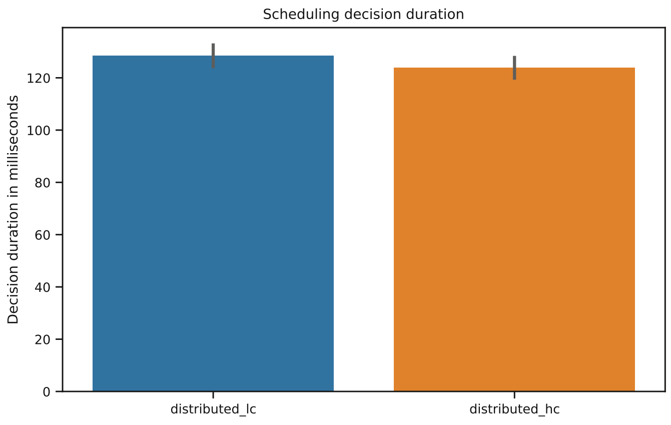


Figure 5.111: Scheduling decision duration for a pod in a distributed architecture, low and high constraint infrastructure

Figure 5.112 illustrates the mean scaling decision duration for an autoscaler. We see that distributed_hc took 57.8ms and a standard deviation of 56.3ms longer for a decision than distributed_lc with 29.5ms and a standard deviation of 44.2ms.

In Figure 5.113, we illustrate the course of up and down scaling throughout the experiments in distributed_lc and distributed_hc. In distributed_hc, we had an intense up scaling right at the beginning, which was then constant for most of the experiments' duration.

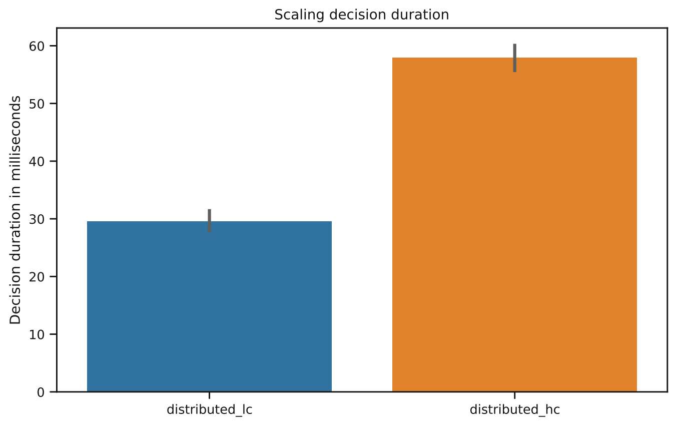


Figure 5.112: Scaling decision duration for a pod in distributed architecture, low and high constraint infrastructure

It reached a maximum of 79 concurrent running replicas. In distributed_lc, there was a significant scale-up only in the second half of the experiments, while the first half was relatively constant. It reached a maximum of 88 concurrent running replicas.

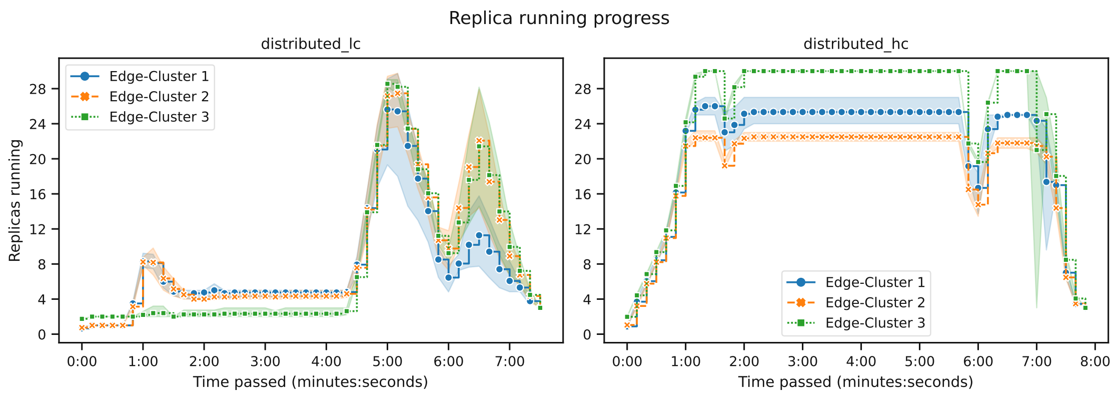


Figure 5.113: Replica running progress in distributed architecture, low and high constraint infrastructure

CHAPTER 6

Recommendation

This chapter includes a discussion and an assessment based on the earlier results. The discussion contains more background information about the results, which is used for the following recommendation. The recommendation is divided into three parts, each from different perspectives: user, platform provider, and impact of infrastructure on architecture. For the first two perspectives, we present the best architecture for our two scenarios, urban sensing and IIoT, which correspond to high and low constraint infrastructure. The parameter sets used for these two perspectives are *hc* for urban sensing and *lc* for IIoT. For the third perspective, we look at the data from the experiments in which we used the parameter set *hc_lc*.

6.1 Discussion

The discussion section is categorized by the corresponding orchestration performance metric. Before we can start with the discussion, some background information is needed.

Firstly, regarding CPU usage, it is essential to mention that the 50% of CPU utilization on weak nodes was artificially included (see Section 5.1.1).

The second insight relates to the global scheduler in a decentralized architecture. At first, the decentralized scheduler only used CPU usage as a deciding tool, similar to the schedulers of the other architectures. However, since the global scheduler works with average CPU values from each zone, the zone-c, with all strong nodes, was chosen every time. Having more function replicas in zone-c allowed for a better CPU usage distribution, and consequently, no other zone was ever chosen. Therefore, we implemented a locality aspect in the global scheduler of the distributed architecture to combat this behavior. Since we implemented only a rudimentary locality awareness and the Cloud did not create user requests, the global scheduler would only schedule function instances to the Cloud when the other two clusters were full. With our user profile (see Figure 5.4), this situation did not occur, and therefore, the Cloud cluster did not have any function instances during the experiments on a decentralized architecture. While there are better outcomes than this, we decided it was better than our first draft and stuck with it.

The following subsections concentrate on why values have behaved differently relative to each other. We give a detailed overview of the results for the recommendations in Section 6.2, Section 6.3, and Section 6.4.

6.1.1 System level

The main influential points in CPU and RAM usage are the number of running replicas, the load distribution of the user workload, and the number of weak and strong nodes in the clusters. The actual data regarding resource usage can be found in Section 5.3.2, Section 5.3.3, Section 5.3.4, and Section 5.3.5.

For the low constraint infrastructure, we see that since the amount of weak nodes is so low, all the clusters had a mean CPU usage of under 50%. The clusters with the highest strong-to-weak ratio had the lowest resource usage. The only exception was the Cloud cluster in the decentralized architecture. The low usage of the Cloud cluster in the decentralized architecture was because no function replicas were running on this cluster. The actual results can be seen in Figure 5.15, Figure 5.25, Figure 5.36, and Figure 5.45.

We want to highlight some aspects regarding resource usage for the control components in the low constraint infrastructure. Firstly, the mean CPU usage of a function replica was directly connected to the amount of running function replicas throughout the experiment. We can see that the function replicas in the centralized architecture had a very low CPU consumption due to the high number and longevity of function replicas. For the other two architectures, the opposite was true. The average CPU usage of a function replica in a low constraint environment is illustrated in Figure 5.19, Figure 5.29, Figure 5.39, and Figure 5.49. Secondly, looking at the resource usage from the load-balancer, the difference between the values in the different architectures was attributable to the fact that the load-balancers in the distributed architecture only had to manage the workload of one client. In contrast, the other architectures had to manage two clients. The slightly higher values between centralized and decentralized came from the zone crossings in decentralized since they create an overhead in decentralized architecture. The average CPU usage of the load-balancer in a low constraint environment is illustrated in Figure 5.21, Figure 5.31, Figure 5.41, and Figure 5.51. The average RAM usage is depicted in Figure 5.22, Figure 5.32, Figure 5.42, and Figure 5.52. Lastly, the mean CPU usage of the schedulers was directly connected to the number of up-scaling requests from the autoscaler. It is vital to mention that in the decentralized architecture, the values of the global and local scheduler in the Cloud cluster are not handled separately. The average CPU usage of a scheduler in a low constraint environment is illustrated in Figure 5.23, Figure 5.33, Figure 5.43, and Figure 5.53.

Since the number of weak nodes is so high in the high constraint infrastructure, we can see its impact on pushing the resource usage of all the architectures around the 50% mark. In contrast to the low constraint infrastructure, the clusters with the lowest strong-to-weak ratio had the lowest resource usage. The only exception was the Cloud since no replicas were deployed on this cluster. The actual results can be seen in Figure 5.5, Figure 5.25, Figure 5.36, and Figure 5.45.

Regarding the resource usage for the control components in the high constraint infrastructure, we had a similar situation as in the low constraint infrastructure. When we look at the mean CPU usage of a function replica, we see that architectures with a high scale amount had a low CPU consumption. The high values of the Edge-Cluster 1 and Edge-Cluster 2 had an additional background. These two clusters had already reached their local maximum of allowed running replicas but still had to burden the workload of one client each. Thereby, these clusters' nodes were overloaded, further increasing CPU usage. The average CPU usage of a function replica in a high constraint environment is illustrated in Figure 5.9, Figure 5.29, Figure 5.39, and Figure 5.49. The same argument from the low constraint infrastructure holds for the resource usage of the load-balancers. The only difference was that the zone transitions in the decentralized architecture had less of an effect here. The average CPU usage of the load-balancer in a high constraint environment is illustrated in Figure 5.11, Figure 5.31, Figure 5.41, and Figure 5.51. The average RAM usage in Figure 5.22, Figure 5.32, Figure 5.42, and Figure 5.52. In the high constraint infrastructure, the CPU usage of the schedulers was also directly connected to the number of scaling requests from the autoscaler. The reason why the values from Edge-Cluster 1 and Edge-Cluster 2 from the distributed architecture were so high goes back to a misconfiguration of the autoscalers in these clusters. As mentioned in 5.2.1, each local autoscaler could scale up to 30 running replicas. However, Edge-Cluster 1 and 2 could not host 30 function replicas in the high constraint infrastructure. As a result, the autoscalers tried in vain to increase the number of replicas, and the schedulers had to undergo the whole scheduling process before declining the scheduling of the function replica. This situation highly increased the CPU usage of the scheduler in Edge-Cluster 1 and Edge-Cluster 2. Despite the misconfiguration, we have gained insights into the importance of resilience to node failures. Dealing with node failures is essential to maintaining an acceptable performance level [56]. Among other things, proactive or reactive behavior is crucial to alleviate this problem, and our framework supports this with, for example, custom dynamic configurable control components. The average CPU usage of a scheduler in a high constraint environment is illustrated in Figure 5.13, Figure 5.33, Figure 5.43, and Figure 5.53.

As part of the *cpu* and *ram* metric, we also collected the resource usage categorized per node type. However, it did not give us enough information to formulate an insight. We assume that data like this would be more useful if a variety of nodes were included or if the initial CPU stress between the nodes was closer together. Furthermore, we did not dive deeper into the average RAM usage of function replicas, schedulers, and autoscalers. The reason regarding the function replica was that since the function was mainly CPU intensive, the RAM usage was negligible. We also did not analyze the scheduler or autoscaler RAM consumption since it was almost identical for every architecture and infrastructure. In addition, we did not analyze the CPU usage of the autoscaler any further, as the main reason for the usage was not the number of scaling operations but the preprocessing of the requests for the scaling decision. Since the preprocessing was done independently of the scaling decision, we excluded it from our recommendation.

6.1.2 Network level

Before starting the discussion about the network throughput, it is essential to mention that the decentralized architecture had three clients, while the other two architectures only had two. Therefore, more requests were sent, so the overall network throughput in the decentralized architecture was the highest on both infrastructures. The overall network throughput, and also the network throughput for each control component can be found in Section 5.3.6, Section 5.3.7, Section 5.3.8, Section 5.3.9, and Section 5.3.10.

The control nodes were the most network-hungry nodes for all architectures in both infrastructures. The network throughput values were mostly low since the number of worker nodes was much higher than the control nodes. This explains the high value outliers in Figure 5.55, Figure 5.61, Figure 5.67, Figure 5.73, and Figure 5.79. When we looked deeper into the network throughput of the control nodes, we found out that the load-balancer had the most network throughput by far. The scheduler and autoscaler had a negligible impact on the network throughput. Furthermore, since the scaling pattern differed for the same architecture on different infrastructures, we concluded that the infrastructure also impacted the controller nodes' network throughput.

The main impact of the network throughput from the load-balancers was the number of user requests they had to handle. In the distributed and decentralized architecture, each load-balancer only had to process one client workload, while the load-balancer from the centralized architecture had two. However, the reason why, in the low constraint infrastructure, the load-balancers from the centralized architecture had the same throughput as the ones in the decentralized architecture was due to zone crossings. These crossings create additional overhead for the decentralized architecture and increase the network throughput since the requests need to be sent to the load-balancer of the other zone. Furthermore, due to the nonexistence of function replica on the Cloud in the decentralized architecture, the load-balancer's depicted throughput was the idle network throughput. The actual average network throughput of the load-balancers are illustrated in Figure 5.59, Figure 5.65, Figure 5.71, Figure 5.77, and Figure 5.83.

The function replicas themselves also had an impactful role in the network throughput. The fewer function replicas were running, or the worse the user requests were distributed, the higher the average network throughput of a function replica was. The actual average network throughput values of function replicas can be seen in Figure 5.58, Figure 5.64, Figure 5.70, Figure 5.76, and Figure 5.82.

The network latency time indicates how long it took to process the request without the execution time of the function. Overloaded function replicas, or worker nodes, greatly impacted the latency values. The values for the low constraint infrastructure were influenced mainly by the artificially added latency values, described in Table 5.5. Since the latency between Edge and Cloud was the biggest, and in the centralized architecture, everything got sent to the Cloud, it made sense that the latency for the centralized architecture was the highest. With the same mindset, it made sense that the distributed architecture had the lowest latency values since there was no inter-cluster communication. For the high constraint infrastructure, a different aspect also had a significant impact in addition to the artificial higher latencies. There, nodes were overloaded with their workload, especially in the distributed and decentralized architecture. In the function replicas, queues started to form for processing the user requests. Since the function did not start executing, even though it was already routed to a function replica, this waiting time was also counted as part of the latency. The distribution of latency values for all architectures and infrastructures can be found in Figure 5.56, Figure 5.62, Figure 5.68, Figure 5.74, and Figure 5.80.

6.1.3 Application level

The collected data regarding *at*, and *rtt* can be found in Section 5.3.11, Section 5.3.12, Section 5.3.13, Section 5.3.14, and Section 5.3.15.

Regarding the *at* metric, there were no significant differences between the infrastructures and architectures. The sent pattern is slightly different across the architectures and infrastructures because the clients created their requests synchronously, and depending on the performance, the waiting time differed. The difference was, however, insignificant since the total number of requests processed was around the same at the end, which can be read from the average application throughput being similar across the infrastructures and architectures. The distributed architecture had such a higher application throughput because it had one more client than the other two architectures. Therefore, in total, more requests were created and processed, thereby resulting in higher throughput. Moreover, it is important to mention that the way this data was processed did not represent the performance of an architecture. Since we only looked at the finished request per second, it was possible that older requests, which just took a long time, finished at the same time as newly started requests. Thereby, a high throughput did not equal a better performance. Moreover, since the distributed architecture had one more client than the other architectures, it further made it difficult to argue using this metric. The average application throughput for each architecture and infrastructure is illustrated in Figure 5.86, Figure 5.88, Figure 5.90. Figure 5.92, and Figure 5.94.

The metric *rtt* is an excellent way to represent the performance of an architecture. The round trip time consists of the function execution time and the latency. A high round trip time can arise when a function replica or a worker node is overloaded or due to high latency.

For the low constraint environment, centralized achieved the best rtt values due to its strong and evenly spread scaling behavior. The distributed architecture was a close second due to its well-executed distribution of user requests. The distribution of rtt values for the low constraint environment is illustrated in Figure 5.87, Figure 5.89, Figure 5.91, and Figure 5.93. The distributed architecture provided the best round-trip time values in the high constraint environment due to its evenly distributed scaling pattern. The centralized architecture could not perform well due to its unevenly spread scaling. The distribution of round trip time values for the high constraint infrastructure is illustrated in Figure 5.85, Figure 5.89, Figure 5.91, and Figure 5.93,

As we have split the rtt values according to the request path, we can see the impact of the latency caused by zone crossings in the decentralized architecture. It is essential to mention that zone crossings created an overhead only in the distributed architecture and not in the centralized architecture. This difference was because the decentralized architecture requires additional communication between the local load-balancers, whereas in the centralized architecture, everything is handled by a global load-balancer. Furthermore, there was an interesting result when we compared the same architecture on different infrastructures. While the centralized and distributed architecture performed visibly worse on the high constraint infrastructure, the decentralized architecture had a similar, if not even better, distribution of rtt values on the high constraint infrastructure than the low constraint one. This difference is probably attributable to the higher and more evenly spread scaling pattern of the decentralized architecture on the high constraint infrastructure.

6.1.4 Orchestration level

The results of the metrics described in this section can be found in Section 5.3.16, Section 5.3.17, Section 5.3.18, Section 5.3.19, and Section 5.3.20.

For the centralized architecture in both infrastructures, the Cloud cluster was the most frequent target for the zone crossings. Since with the centralized architecture, each request first went to the load-balancer placed in the Cloud. Leaving the Cloud again would add some additional latency to the request. In contrast to the centralized architecture, the decentralized architecture had relatively few zone transitions. However, since they had rather different replica scaling patterns on the two infrastructures, we saw that a high and steady replica count led to fewer zone crossings. As no function replicas and clients were running in the Cloud, there were no zone crossings to or from the Cloud. Regarding the distributed architecture, since there is no inter-zone communication, there were zero zone crossings with the distributed architecture. The results of the *zc* metric are illustrated in Figure 5.95, Figure 5.99, Figure 5.103, and Figure 5.107.

In order to illustrate the *tos* metric, we looked at the state of the clusters regarding running replicas during the experiments. The results are illustrated in Figure 5.98, Figure 5.102, Figure 5.106, Figure 5.110, and Figure 5.113.

For the high constraint infrastructure, the centralized architecture had, on average, the most running replicas. Theoretically, the distributed architecture could have reached a higher replica count. While the autoscalers in Edge-Cluster 1 and 2 reached their local resource maximum, the autoscaler in Edge-Cluster 3 was limited by our scaling configuration. For all three architectures, the scaling pattern was relatively similar in the high constraint environment. The scaling pattern differed between centralized, distributed, and decentralized architecture in the low constraint environment. Here, the centralized architecture scaled aggressively early, while the other two architectures only scaled at the time of the peaks of the user profile. The centralized architecture also reached the maximum of concurrent running function replicas, so it could have been scaled up even higher. With that information, the centralized architecture might tackle better unforeseen workloads, but the others are more resource-efficient.

We found that, on average, clusters in the high constraint environment had more replicas running than in the low constraint environment. For the centralized architecture, we saw that in the high constraint environment, most of the upscaling happened in the Cloud. In contrast, in the low constraint environment, the scaling was more spread. We estimate that due to the sheer resource difference between the Cloud and the other clusters in the high constraint infrastructure, the Cloud was a more attractive schedule target than the rest. This difference does not dominate the low constraint infrastructure, allowing for more spread scheduling. While both infrastructures had a similar scaling pattern, it is vital to mention that the maximum amount of replica was reached in the low constraint environment, limiting the scaling. So, the scaling pattern may have differed with a higher maximum replica count. Furthermore, when we look at the scaling pattern in decentralized and distributed architectures, the different infrastructures alone have caused different scaling patterns. The clusters reached their limit relatively early for the distributed architecture in the high constraint environment. In contrast, in the low constraint environment, a significant scaling only occurred at the end and held on short. For most experiments, the distributed architecture in the low constraint environment handled the workload with a meager number of function replicas. The decentralized architecture had a similar scaling pattern for both infrastructures as the distributed architecture. Furthermore, for both infrastructures, the decentralized architecture had, across the three architectures, the lowest amount of running replicas.

In general, when we compared the number of average running replicas together with latency and round trip time values, we were able to see that a high replica count most possibly allowed for better workload distribution, causing fewer replica or node overloading and, therefore, better round trip time and latency values. Furthermore, it also helped reduce the network throughput of a function replica.

The *schedd* and *scaled* metrics can significantly impact the system's overall performance. A long scaling or scheduling decision can lead to a significant degradation of the overall performance. For both high and low constraint environments, the scheduling decision was significantly longer for the decentralized architecture than for the other two. The longer decision time was because, in a decentralized architecture, a function replica had to be processed by two schedulers before it could be scheduled. The scheduling duration decisions are depicted in Figure 5.96, Figure 5.100, Figure 5.104, Figure 5.108, and Figure 5.111. For the autoscalers, in our case, the most time-consuming task was the preprocessing of processed requests to calculate the desired number of function replicas.

The duration of a decision, for both *schedd* and *scaled*, was, in our case, too short to have a significant impact on performance. The scaling durations are illustrated in Figure 5.97, Figure 5.101, Figure 5.105, Figure 5.109, and Figure 5.112.

The following sections contain the orchestration architecture recommendations from different perspectives. The recommendations are based on the findings from Section 6.1.

6.2 User Perspective

This section only focuses on the *rtt* metric since it is most visible to the user and clearly illustrates how fast a request is processed. Although latency also plays a vital role in the duration of request processing, we omit the *lat* metric since the latency is indirectly included in the rtt values. Table 6.1 contains the aggregated rtt values for all three architectures using both infrastructures.

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%	Difference to target value in %
centralized	low	0.55s	0.07s	0.55s-0.56s	85%
decentralized	low	2.55s	3.72s	2.51s-2.60s	751%
distributed	low	0.78s	1.10s	0.77s-0.79s	159%
centralized	high	2.51s	2.71s	2.48s-2.55s	737%
decentralized	high	2.10s	3.02s	2.06s-2.14s	601%
distributed	high	1.68s	1.86s	1.66s-1.70s	461%

Table 6.1: Aggregated round trip time values for all three architectures using *lc* and *hc* parameter sets

For the IIoT scenario, based on our collected data, the centralized architecture performed, with an aggregated mean value of 0.55s and low spread the best. The distributed architecture was also an acceptable contender, while the decentralized architecture is not recommended due to its high aggregated mean value of 2.55s and very high aggregated standard deviation of 3.7s. As we can see from the last column, no architecture reached our target duration of 0.3s. With that comparison, we can see that centralized is the clear winner with an increase of 86%, while decentralized is the worst by far with an increase of 751%.

For the urban sensing scenario, we recommend the distributed architecture. In contrast to the other scenario, distributed architecture significantly outperformed centralized and decentralized architecture. While the percentage increase was very high for all three architectures, the distribution was 461% (1.68s), by far the lowest. Also, its aggregated confidence interval is around 0.5s smaller than the other two architectures.

With that information, both the centralized and distributed architecture are favorable from the user perspective, depending on the infrastructure.

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%
centralized	low	19	11	18-19
decentralized	low	3	4	3-4
distributed	low	8	7	7-8
centralized	high	21	14	20-22
decentralized	high	7	7	7-8
distributed	high	18	8	18-19

Table 6.2: Aggregated number of replicas running over all clusters for all three architectures using *lc* and *hc* parameter sets

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%
centralized	low	25.31%	19.31%	24.78%-25.85%
decentralized	low	25.53%	22.93%	24.92%-26.13%
distributed	low	30.30%	24.71%	29.70%-30.91%
centralized	high	39.91%	22.34%	39.28%-40.55%
decentralized	high	34.97%	24.05%	34.33%-35.60%
distributed	high	49.69%	22.90%	49.10%-50.27%

Table 6.3: Aggregated CPU usage over all clusters for all three architectures using *lc* and *hc* parameter sets

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%
centralized	low	774	350	610-938
decentralized	low	472	146	366-577
distributed	low	0	0	0
centralized	high	951	541	698-1205
decentralized	high	221	53	182-259
distributed	high	0	0	0

Table 6.4: Aggregated amount of zone crossings for all three architectures using *lc* and *hc* parameter sets

6.3 Platform Provider Perspective

In this section, we mainly focus on the hypothetical operating cost. For this, we consider *ram*, *cpu*, *net*, *zc*, and *tos* metric.

When we bring these metrics together, we can select the decentralized architecture as the cheapest for both scenarios. The main argument of this decision was the *tos* metric since the amount of running replicas in the decentralized architecture was much lower than the other two architectures. As we can see in Table 6.2, the aggregated number of replicas running on average was 3 in the low constraint and 7 in the high constraint infrastructure, by far the lowest. For the system level metrics, we concentrate mainly on *cpu* since both the control component algorithms and application were predominantly CPU intensive, and lower CPU usages allows for more applications to be used. From Table 6.3, we see that decentralized shares the spot with the centralized architecture in the low constraint infrastructure with the lowest aggregated mean CPU usage of 25%. However, since the decentralized architecture in the highly restricted environment with 5% had the lowest CPU usage, the resource usage also speaks in favor of the decentralized architecture as the most low-cost. Metric *net* was not that expressive, as all three architectures mostly had similar values. Except that, in the decentralized architecture, the net usage on the Cloud cluster and some of its components was very low due to the lack of function replicas on the cluster. The only metric that could massively affect the running cost so that the distributed architecture would be the cheapest is *zc* since zone crossings are associated with a price [1]. However, even in this case, although distributed had no zone crossings in general, the number of crossings in decentralized was relatively low for the provided workload, see Table 6.4.

6.4 Impact of the Infrastructure on the Architecture

Most of the metrics for each architecture were unaffected by the infrastructure. The number of zone crossings was relatively similar for the centralized architecture in both infrastructures. The same holds for the decentralized architecture. Furthermore, for all three architectures, the infrastructure did not significantly impact the network throughput. Similarly, for each architecture, the collected metrics on the container level did not show any significant impact from the infrastructure. In general, the most significant differences were on the *cpu*, *ram*, *lat*, *rtt*, and *tos* metric.

Regarding the *cpu* and *ram* metric, the high constraint environment had a higher average CPU and RAM usage for all three architectures. Similar to the section above, we only look closer at the CPU usage since the application and algorithms used in the experiment were mostly CPU intensive. The average CPU usage in the high constraint infrastructure was higher because the weak nodes, which start with a CPU load of 50%, outnumber the strong nodes. As depicted in Table 6.5, the highest difference was 21% in the distributed architecture, and the lowest was in the decentralized architecture with 10%.

The *tos* metric was one where we could identify a clear impact from the infrastructure. We saw a more spread scaling behavior for the centralized architecture in the low constraint infrastructure than in the high constraint one. Furthermore, the autoscaler in the high constraint environment started to scale up relatively early. In contrast, in the low constraint environment, only at the big spikes and the end of the user profile did the autoscaler decide to scale up rapidly. These difference scaling patterns substantially impacted the *lat* and *rtt* metrics. According to the average number of replicas running in centralized and decentralized architectures, both infrastructures had

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%
centralized	low	25.57%	18.73%	25.03%-26.10%
centralized	high	38.76%	22.06%	38.14%-39.38%
decentralized	low	24.09%	22.22%	23.50%-24.67%
decentralized	high	34.92%	24.45%	34.28%-35.57%
distributed	low	29.46%	24.22%	28.87%-30.05%
distributed	high	50.02%	22.14%	49.44%-50.59%

Table 6.5: Aggregated CPU usage over all clusters for all three architectures using *hc_lc* parameter set

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%
centralized	low	21	11	20-22
centralized	high	18	11	18-19
decentralized	low	4	5	3-4
decentralized	high	7	8	7-8
distributed	low	8	7	8-9
distributed	high	21	8	20-21

Table 6.6: Aggregated number of replicas running over all clusters for all three architectures using hc_lc parameter sets

a similar number. The mean number of running replicas in distributed was in the high constraint infrastructure nearly three times as high as in the low constraint one, as shown in Table 6.6.

Since the latency values between clusters were artificially higher in the high constraint infrastructure than the low constraint infrastructure, it was no surprise that the latency values were lower in the low constraint infrastructures, see Table 6.7. The most significant difference in latency values was in the distributed architecture with 205ms, and the lowest difference with 162ms was in the decentralized architecture. The latency values in low constraint infrastructures were mainly influenced by the artificially configured latency values between Edge Clusters and the Cloud. At the same time, in the high constraint environment, the impact of overloaded function replicas and queuing of user requests was more prominent, especially for the distributed architecture.

For the *rtt* metric, we had a similar result as for the *lat* metric. Here, the centralized and distributed architecture also performed significantly better in the low constraint infrastructure, while the decentralized architecture performed equally in both infrastructures. We show the aggregated values in Table 6.8. The centralized architecture showed the most prominent difference with 1.26s, and the decentralized architecture showed the lowest difference with just 0.15s.

Architecture	Constraint Level	Mean	Std	Confidence Interval 95%
centralized	low	130.99ms	23.37ms	132.68ms-133.31ms
centralized	high	294.27ms	190.28ms	291.62ms-296.93ms
decentralized	low	138.42ms	447.30ms	130.73ms-142.10ms
decentralized	high	300.78ms	451.34ms	295.08ms-306.49ms
distributed	low	45.77ms	7.60ms	45.69ms-45.85ms
distributed	high	250.54ms	261.68ms	247.84ms-253.24ms

Table 6.7: Aggregated latency values for all three architectures using hc_lc parameter set

Architecture	Constrain level	Mean	Std	Confidence Interval 95%	Difference to target value in %
centralized	low	0.56s	0.08s	0.55s-0.56s	85%
centralized	high	1.82s	2.04s	1.79s-1.85s	507%
decentralized	low	1.92s	3.02s	1.88s-1.95s	539%
decentralized	high	2.07s	2.90s	2.03s-2.11s	590%
distributed	low	0.68s	1.01s	0.67s-0.69s	127%
distributed	high	1.69s	1.81s	1.68s-1.71s	465%

Table 6.8: Aggregated round trip time values for all three architectures using hc_lc parameter set

To wrap this section up, the decentralized architecture was the least affected by the infrastructure. For all metrics shown, the decentralized architecture had the lowest difference in value for all values, and for *rtt*, the values remained quite similar. While the centralized architecture showed the most significant difference in performance in *rtt*, the distributed architecture was the most affected by infrastructure for all other metrics shown.

CHAPTER 7

Conclusion

Edge computing has been introduced as a new paradigm handling the QoS and QoE of new applications, such as autonomous vehicles or augmented reality. With it came significant application deployment challenges since edge computing needs to handle, among other things, heterogeneous hardware and network technologies. For example, before the edge computing paradigm, developers could choose the hardware for their application deployment. However, this is challenging in edge computing due to the heterogeneous nature of the paradigm. In order to combat these deployment challenges, a different paradigm, serverless computing, came to life, where applications are implemented as stateless functions. While this paradigm makes it easier for developers to deploy applications, the service providers have to face the responsibility of resource management. Part of resource management is optimal resource orchestration. Providing appropriate scheduling and scaling strategies to manage the resources effectively without hurting predefined SLOs is essential. Current literature presents multiple resource orchestration architectures, implying that the best one has yet to be found. There are a lot of different aspects which can play a role in finding an optimal architecture. Since a heterogeneous infrastructure is an essential characteristic of edge-cloud computing, the architecture must be able to perform in different scenarios. Secondly, there are many metrics, following the taxonomy of [3], which reflect the performance of an edge-cloud system, making it a difficult task to evaluate an orchestration architecture approach. Lastly, the optimal orchestration architecture depends on the perspective. Different perspectives, such as those of the user and the platform provider, are likely to have different interests. All these make it challenging to make an optimal decision for an orchestration architecture for an application.

Our work intends to improve this aspect of serverless edge computing by providing a framework capable of recommending orchestration architectures for different infrastructures. We first perform an initial analysis of existing orchestration architectures used in serverless edge computing. Based on this, our recommendation contains three architectures: centralized, decentralized, and distributed. As heterogeneity is an essential part of edge computing, we also analyze the different scenarios in which edge computing systems are utilized to be closer to the real world. As a result, we test each architecture in an urban sensing and IIoT scenario corresponding to a low and high constraint infrastructure. Next, we research meaningful performance metrics on which to base our recommendation. We classify the performance metrics into four categories: system, network, application, and orchestration. To conduct the experiments, we use our framework, which extends the Galileo benchmarking tool. The extension consists of deploying custom clusters with specified constraints, implementing custom control components appropriate to the operating principles of a chosen orchestration architecture, and extending preexisting metric data processing. We base our framework on a system model capable of modeling edge-cloud systems' heterogeneous infrastructures and orchestration architectures. Utilizing our framework, we create a recommendation from three perspectives: user, platform provider, and infrastructure impact of the architectures.

The results show that, from the user's perspective, centralized architecture is recommended in an IIoT scenario, while decentralized architecture is recommended in an urban sensing scenario. From the other two perspectives, the decentralized architecture outperformed the other two architectures in both scenarios.

7.1 Research Questions

In this section, the answers to the research questions addressed in this thesis are summarized in order to highlight the key findings and contributions of the work.

RQ1 How can we describe and categorize heterogeneous edge-cloud systems and orchestration architectures?

Since there are numerous edge-cloud systems, each with heterogeneous hardware, networks, and more, a categorization is beneficial to create some structure. To create a meaningful recommendation, we need to find edge cloud systems that are common and heterogeneous for our experiments. Based on our literature research, we have decided on the urban sensing and IIoT scenarios. Urban sensing is a component of the Smart City movement, which aims to collect environmental data for the use of citizens and government, and the IIoT is a crucial element of Industry 4.0. Looking further into their characteristics, we found significant differences in their heterogeneous network and hardware systems and categorized them based on these differences. Urban sensing was more constrained than the IIoT scenario regarding hardware and network resources. For example, an urban sensing scenario often consists of many sensors running on single board computers, while using dedicated edge data centers or compute clusters can be observed in IIoT scenarios [60]. Based on this, we created two different infrastructures that differ in the amount of hardware and network constraints. The high constraint infrastructure illustrates the urban sensing scenario, while the low constraint infrastructure depicts the IIoT scenario. To keep the heterogeneity aspect, we decided for the network heterogeneity different latency values between edge clusters. As far as hardware heterogeneity is concerned, we have introduced two types of nodes, which differ in the size of CPU and RAM resources. The low constraint infrastructure contains more resource-stronger nodes, while the high constraint infrastructure contains more resource-weaker nodes.

Since more than this type of categorization is needed to use as a basis for our framework, we also created a system model capable of describing the orchestration of edge-cloud systems for different architectures and heterogeneous infrastructures. It can describe the different hardware and network constraints and also covers the functionality of control components, which is vital for modeling different orchestration architectures. This system model served as the basis for our framework. Nevertheless, the system model can be extended towards different use cases. One example would be adding energy consumption information to the nodes, as sensors, operated with batteries, are often used in urban sensing. Battery consumption can, therefore, play a significant role in this scenario.

RQ2 What KPIs are appropriate to evaluate an orchestration architecture?

To create a meaningful recommendation, expressive KPIs are vital. To find these expressive KPIs, we conducted a literature research. Since we found a lot of different metrics, we decided to pick metrics based on the possible interests of our perspectives, the type of data collection capabilities of our framework, and general occurrence in literature. We then divided the chosen metrics into four different categories: system level, network level, application level, and orchestration level. The first level contains the metrics describing the resource usage of singular components and nodes. We looked at the percentage of CPU and RAM usage. For the network level, we decided to collect the network throughput of nodes and single components and the system latency of the requests. The application level describes metrics influenced by the used application and consists of the round trip time of user requests and the application throughput. Finally, the orchestration level contains metrics referring to the orchestration actions. This level includes the number of times a user request was processed in a zone different from the user's location, the duration of a scheduling and scaling decision, and the general amount of scaling done. While there are more metrics than we covered out there, the chosen metrics should be enough to judge the performance of an orchestration strategy.

RQ3 How can we create a system that allows and illustrates the differences between orchestration architectures and heterogeneous infrastructures and can make recommendations for those?

Using the KPIs from RQ2 and the categorization and system model from RQ1, we can develop a framework to find the most optimum orchestration architecture based on predefined QoS metrics. Our framework is an extension of the end-to-end benchmark framework Galileo. We extended some of the original framework's features and added new ones to ensure we covered all aspects of our newly created system model. We incorporated dynamically creating VMs, which embody the nodes of the edge-cloud clusters. The framework then entirely creates a K3S cluster from the VMs automatically. Then, control and metric collection components get deployed onto the cluster. We extended the existing autoscaler and load-balancer implementation of the original framework to fulfill the requirements of the work behavior of the different orchestration architectures. Since Galileo provided no custom scheduler, we implemented one for each architect. The framework can deploy any component provided it is deployable on a Kubernetes cluster. Next, we also integrated the network traffic shaping tool tc to create custom network latency between clusters. Lastly, while the framework collected all the raw metric data we needed, we still had to do some preprocessing before we used the collected data for our chosen metrics. To summarize, we have extended the framework Galileo to include the following features: dynamically building K3S clusters with fully configurable VMs, extending existing control components and deploying custom components, integrating the network traffic shaping tool TC, and preprocessing the collected metrics data.

To evaluate our framework and our chosen KPIs, we perform a use case study in which we evaluate the different orchestration architectures, focusing on the orchestration components and application behavior in different scenarios. From the evaluation of the data collected with the framework, we then conducted a recommendation using the earlier described KPIs based on three different perspectives: user, platform provider, and the impact of different infrastructures on the architectures. We only considered the round trip time from the user perspective and concluded that the centralized architecture performed best in the IIoT scenario, with an aggregated mean value of 0.55s. In the urban sensing scenario, the lowest aggregated mean value was in the distributed architecture, with 1.68s. From the perspective of a platform provider, the best choice would be the decentralized architecture for both scenarios due to its low aggregated number of running replicas of 3 in low constraint and 7 in high constraint infrastructure. Moreover, the mean CPU usage of the nodes was the lowest in the decentralized architecture, and the number of zone crossings was low in perspective of the workload. For the third perspective, we also recommend decentralized architecture since it was least affected by the infrastructure of all architectures. For all metrics considered by the perspective, the difference in values between the high and low constraint infrastructures was the lowest.

7.2 Future Work

During this thesis, we identified areas for improvement and further used cases for our framework. The following section introduces possible future research areas.

- For our recommendation, we did not consider the *schedd* and *scaled* metrics since, in our case, the implementations were too simple to have a significant impact. However, the impact of these two metrics should not be underestimated, especially due to possible cascading delays with, for example, the reconciliation interval. Imagine a scenario where the scheduling duration of one replica is 2 seconds, and the reconciliation interval of the autoscaler is five seconds. If the autoscaler creates ten new replicas, a problem could occur where the reconciliation interval is over before all replicas are scheduled since it would take 20 seconds to schedule all ten replicas. In this situation, a new scaling decision would be made without fully considering the old decision. This and similar problems present an ideal opportunity to utilize our framework and investigate the impact of more sophisticated scheduling and scaling algorithms.

- Since our control components use configurable parameters, it would be an interesting use case for our framework to investigate further the effects of the control components' parameters on the system. For example, the reconcile interval parameter alone can impact the scaling behavior [23].

- In our experiments, most of the network throughput was created by the load-balancers, as described in 6.1.2. Experimenting with placement and decision strategies of load-balancers can significantly impact response times and performance [52]. The framework provides an opportunity to investigate different load-balancing strategies and gain insights into different strategies' strengths and weaknesses.

- While we already covered some interesting performance metrics, our recommendation can be extended by including additional QoS levels. For example, [3] introduces a comprehensive taxonomy of performance metrics for evaluating the performance of edge-cloud systems, which can be taken as inspiration for additional metrics. Some additional metrics could be energy consumption, reliability, availability, and more.

- Since industrial edge-cloud systems use commercial cloud providers [53], porting our framework to an existing provider, such as AWS, Azure, and GCP, would improve the data and results' relevance.

- In the current state of our framework, we cannot differentiate between function replicas processing user requests and idle replicas. This results in the problem that sometimes, client requests are broken during a scale-down since the function replica was shut down during processing. For future work, a sophisticated strategy must be found to handle this situation.

- Finally, the ultimate objective of our framework is to turn it into a recommendation system so that when a user provides different metrics, architectures, and infrastructures, the system provides the most optimal solution.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Amazon. Amazon ec2 on-demand pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`, 2016. Accessed 23-11-2023.

[2] Ansible. `https://docs.ansible.com/ansible/latest/index.html`, 2021. Accessed 23-06-2023.

[3] M. S. Aslanpour, S. S. Gill, and A. N. Toosi. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet of Things*, 12:100273, 2020.

[4] F. Aznoli and N. J. Navimipour. Cloud services recommendation: Reviewing the recent advances and suggesting the future research directions. *Journal of Network and Computer Applications*, 77:73–86, 2017.

[5] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[6] L. Baresi, D. Y. X. Hu, G. Quattrocchi, and L. Terracciano. Neptune: Network- and gpu-aware management of serverless functions at the edge. In *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '22, page 144–155, New York, NY, USA, 2022. Association for Computing Machinery.

[7] L. Baresi and G. Quattrocchi. Paps: A serverless platform for edge computing infrastructures. *Frontiers in Sustainable Cities*, 3, 2021.

[8] S. Böhm and G. Wirtz. A quantitative evaluation approach for edge orchestration strategies. In S. Dustdar, editor, *Service-Oriented Computing*, pages 127–147, Cham, 2020. Springer International Publishing.

[9] G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, and C. A. da Costa. Serverless computing for internet of things: A systematic literature review. *Future Generation Computer Systems*, 128:299–316, 2022.

[10] C. E. Catlett, P. H. Beckman, R. Sankaran, and K. K. Galvin. Array of things: A scientific research instrument in the public way: Platform design and early lessons learned. SCOPE ’17, page 26–33, New York, NY, USA, 2017. Association for Computing Machinery.

[11] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, and Q. Zhang. Edge computing in iot-based manufacturing. *IEEE Communications Magazine*, 56(9):103–109, 2018.

[12] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo. Dfaas: Decentralized function-as-a-service for federated edge computing. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 1–4, 2021.

[13] B. Costa, J. Bachiega, L. R. de Carvalho, and A. P. F. Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Comput. Surv.*, 55(2), jan 2022.

[14] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.

[15] Docker. `https://www.docker.com/`, May 2022. Accessed 23-06-2023.

[16] Edgerun. `https://github.com/edgerun/faas-optimizations/`, 2023. Accessed 23-06-2023.

[17] Edgerun. `https://github.com/edgerun/go-load-balancer/`, 2023. Accessed 23-06-2023.

[18] Edgerun. `https://github.com/edgerun/telemd/`, 2023. Accessed 23-06-2023.

[19] Edgerun. `https://github.com/edgerun/edge-chaos`, 2023. Accessed 23-06-2023.

[20] fission. fission: Fast and simple serverless functions for kubernetes. `https://github.com/fission/fission`, Mar 2023. Accessed 07-03-2023.

[21] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[22] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao. Concurrent container scheduling on heterogeneous clusters with multi-resource constraints. *Future Generation Computer Systems*, 102:562–573, 2020.

[23] C.-K. Huang and G. Pierre. AdapPF: Self-Adaptive Scrape Interval for Monitoring in Geo-Distributed Cluster Federations. In *ISCC 2023 - 28th IEEE Symposium on Computers and Communications*, pages 1–7, Tunis, Tunisia, July 2023. IEEE, IEEE.

[24] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen. Function delivery network: Extending serverless computing for heterogeneous platforms. *Software: Practice and Experience*, 51(9):1936–1963, 2021.

[25] K3s. `https://k3s.io/`, 2023. Accessed 23-06-2023.

[26] K3s-io. `https://github.com/k3s-io/k3s-ansible`, Jan 2022. Accessed 23-06-2023.

[27] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[28] W. Kim and I. Jung. Simulator for interactive and effective organization of things in edge cluster computing. *Sensors*, 21(8), 2021.

[29] Knative. `https://github.com/knative/`, Mar 2023. Accessed 07-03-2023.

[30] H. Kokkonen, L. Lovén, N. H. Motlagh, A. Kumar, J. Partala, T. Nguyen, V. C. Pujol, P. Kostakos, T. Leppänen, A. González-Gil, E. Sola, I. Angulo, M. Liyanage, M. Bennis, S. Tarkoma, S. Dustdar, S. Pirttikangas, and J. Riekki. Autonomy and intelligence in the computing continuum: Challenges, enablers, and future directions for orchestration, 2022.

[31] Kubeless. kubeless: Kubernetes native serverless framework. `https://github.com/vmware-archive/kubeless`, Dec 2021. Accessed 07-03-2023.

[32] Kubernetes. Horizontal pod autoscaling. `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/`. Accessed 23-11-2023.

[33] Kubernetes. Horizontal pod autoscaling. `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`. Accessed 23-06-2023.

[34] Kubernetes. Kubernetes. `https://kubernetes.io/docs/home/`, Oct 2022. Accessed 24-02-2023.

[35] H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450, 2018.

[36] Z. Li, R. Chard, Y. Babuji, B. Galewsky, T. J. Skluzacek, K. Nagaitsev, A. Woodard, B. Blaiszik, J. Bryan, D. S. Katz, I. Foster, and K. Chard. uncx: Federated function as a service for science. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4948–4963, 2022.

[37] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo. The serverless computing survey: A technical primer for design architecture. 54(10s), sep 2022.

[38] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169, 2018.

[39] O. Ltd. `https://www.openfaas.com/`, 2023. Accessed 07-03-2023.

[40] R. P. Ltd. Buy a raspberry pi 3 model b – raspberry pi. `https://www.raspberrypi.com/products/raspberry-pi-3-model-b/`. Accessed 23-06-2023.

[41] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi. Resource scheduling in edge computing: A survey. *IEEE Communications Surveys Tutorials*, 23(4):2131–2165, 2021.

[42] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169, 2017.

[43] R. Mahmud, R. Kotagiri, and R. Buyya. *Fog Computing: A Taxonomy, Survey and Future Directions*, pages 103–130. Springer Singapore, Singapore, 2018.

[44] A. Mampage, S. Karunasekera, and R. Buyya. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Comput. Surv.*, 54(11s), sep 2022.

[45] X. Masip, E. Marín, J. Garcia, and S. Sànchez. *Collaborative Mechanism for Hybrid Fog-Cloud Scenarios*, chapter 2, pages 7–60. John Wiley Sons, Ltd, 2020.

[46] S. K. Mohanty, G. Premsankar, M. Di Francesco, et al. An evaluation of open source serverless computing frameworks. *CloudCom*, 2018:115–120, 2018.

[47] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, Oct. 2018. USENIX Association.

[48] S. Nastic, P. Raith, A. Furutanpey, T. Pusztai, and S. Dustdar. A serverless computing fabric for edge amp; cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–12, Los Alamitos, CA, USA, dec 2022. IEEE Computer Society.

[49] OpenSSH. `https://www.openssh.com/`, 2023. Accessed 23-06-2023.

[50] OpenWhisk. `https://openwhisk.apache.org/`, 2016. Accessed 24-02-2023.

[51] A. Palade, A. Kazmi, and S. Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642-939X, pages 206–211, 2019.

[52] J. Palecek. Improving serverless edge computing for network bound workloads. 2022.

[53] J. Pan and J. McElhannon. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449, 2018.

[54] T. Pfandzelter and D. Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24, 2020.

[55] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas. Joint service placement and request routing in multi-cell mobile edge computing networks, 2019.

[56] V. Prokhorenko and M. Ali Babar. Architectural resilience in cloud, fog and edge systems: A survey. *IEEE Access*, 8:28078–28095, 2020.

[57] P. Raith and S. Dustdar. Edge intelligence as a service. In *2021 IEEE International Conference on Services Computing (SCC)*, pages 252–262, 2021.

[58] P. Raith, T. Rausch, S. Dustdar, F. Rossi, V. Cardellini, and R. Ranjan. Mobility-aware serverless function adaptations across the edge-cloud continuum. In *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, pages 123–132, 2022.

[59] P. Raith, T. Rausch, P. Prüller, A. Furutanpey, and S. Dustdar. An end-to-end framework for benchmarking edge-cloud cluster management techniques. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–28, 2022.

[60] 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, June 2020.

[61] T. Rausch, P. Raith, P. Pillai, and S. Dustdar. A system for operating energy-aware cloudlets: Demo. SEC '19, page 307–309, New York, NY, USA, 2019. Association for Computing Machinery.

[62] T. Rausch, A. Rashed, and S. Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems*, 114:259–271, 2021.

[63] Z. Rejiba and J. Chamanara. Custom scheduling in kubernetes: A survey on common problems and solution approaches. *ACM Comput. Surv.*, 55(7), dec 2022.

[64] M. A. Rodriguez and R. Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5):698–719, 2019.

[65] R. Singh and S. S. Gill. Edge ai: A survey. *Internet of Things and Cyber-Physical Systems*, 3:71–92, 2023.

[66] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, and E. Chang. Cloud service selection: State-of-the-art and future research directions. *Journal of Network and Computer Applications*, 45:134–150, 2014.

[67] S. Taherizadeh, V. Stankovski, and M. Grobelnik. A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. *Sensors*, 18(9), 2018.

[68] B. Wang, A. Ali-Eldin, and P. Shenoy. Lass: Running latency sensitive serverless computations at the edge. HPDC '21, page 239–251, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Z. Wang, M. Goudarzi, J. Aryal, and R. Buyya. Container orchestration in edge and fog computing environments for real-time iot applications. In R. Buyya, S. M. Hernandez, R. M. R. Kovvur, and T. H. Sarma, editors, *Computational Intelligence and Data Analytics*, pages 1–21, Singapore, 2023. Springer Nature Singapore.

[70] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[71] M. Yu, T. Cao, W. Wang, and R. Chen. Following the data, not the function: Rethinking function orchestration in serverless computing, 2021.

[72] Z. Yu, J. Wang, Q. Qi, J. Liao, and J. Xu. Boundless application and resource based on container technology. In S. Liu, B. Tekinerdogan, M. Aoyama, and L.-J. Zhang, editors, *Edge Computing – EDGE 2018*, pages 34–48, Cham, 2018. Springer International Publishing.

[73] X. Zhou, R. Canady, S. Bao, and A. Gokhale. Cost-effective hardware accelerator recommendation for edge computing. In *HotEdge*, 2020.

[74] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proceedings of the IEEE*, 107(8):1738–1762, 2019.