TU WIEN Informatics

Simulation-Driven Bootstrapping of Edge-Cloud Autoscaler Parameters

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

David Rainer, BSc.
Matrikelnummer 51850574

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 Raith

Wien, 27. März 2025

David Rainer | Schahram Dustdar

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

TU WIEN Informatics

Simulation-Driven Bootstrapping of Edge-Cloud Autoscaler Parameters

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

David Rainer, BSc.
Registration Number 51850574

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 Raith

Vienna, March 27, 2025

David Rainer | Schahram Dustdar

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

Erklärung zur Verfassung der Arbeit

David Rainer, BSc.

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

Wien, 27. März 2025

David Rainer

Acknowledgements

Thank you Mom and Dad, for always believing in me and always supporting me in my pursuits every step of the way. I love you.

Thank you to my advisor Schahram Dustdar and my co-supervisor Philipp Raith, for providing all the help you did throughout the whole process of putting the thesis together. I could not have asked for better support on this journey.

Finally, thank you to my colleagues at work, who always did their best to motivate and enable me to finish my studies. You guys rock.

Kurzfassung

Edge Computing behebt die Bandbreiten- und Latenzprobleme der Cloud und ermöglicht Echtzeitverarbeitung für Anwendungsfälle wie Internet of Things (IoT) und autonomes Fahren. Der Paradigmenwechsel bringt die Notwendigkeit, Orchestrierungsmechanismen der Cloud an das Edgesetting anzupassen. Dies beinhaltet Autoscaler, die Komponenten, welche dafür verantwortlich sind, genutzte Ressourcen je nach Bedarf zu erhöhen oder zu verringern. Eine der großen Herausforderungen, die die breite Akzeptanz moderner Edge-Cloud-Autoscaling-Lösungen behindern, ist die Notwendigkeit von statischer Konfiguration. Beispielsweise kann diese aus komplex aggregierten Granzwerten bestehen, die vorab schwer zu bestimmen sein können. Diese Arbeit untersucht Möglichkeiten zur automatischen Optimierung der Autoscaling-Konfiguration vor dem Deployment mithilfe einer Simulation der Zielinfrastruktur. Zu diesem Zweck werden Leistungsmetriken der Orchestrierung untersucht und acht repräsentative Key Performance Indicators durch ein korrelationsgraphenbasiertes Verfahren ausgewählt. Darüber hinaus wird eine Methode zur Aggregation dieser zu einem Qualitätsscore präsentiert. Eine Erklärung wird gegeben, warum die vorgeschlagene Qualitätsdefinition nur eine von unzähligen gültigen Optionen darstellt. Sechs metaheuristische Optimierungsalgorithmen werden an die vorliegende Aufgabe angepasst. Diese werden anschließend anhand eines Benchmark-Szenarios eines kleinen Smart-City-Edge-Cloud-Deployments miteinander verglichen. Die Ergebnisse zeigen, dass die Single-Objective-Ansätze Particle Swarm Optimization (PSO) und Artificial Bee Colony (ABC) am besten abschneiden. PSO erzielte eine konsistente Verbesserung des Qualitätswerts um etwa 7,1 % im Vergleich zur gewählten Baseline bei geringem Rechenaufwand. ABC erzielte eine durchschnittliche Verbesserung des Qualitätswerts um etwa 8,6 % und in einigen Fällen bis zu 19,4 %, benötigte dafür aber unverhältnismäßig mehr Ressourcen. NSGA-II, der untersuchte Multi-Objective-Ansatz, schnitt unterdurchschnittlich ab. Die Schwierigkeit, eine repräsentative Pareto-Front zu erstellen, deutet darauf hin, dass die Komplexität des Objective-Raums eine Herausforderung darstellt. Die Robustheit der optimierten Konfigurationen unter verschiedenen Deployment-Szenarien wird analysiert. Während die optimierten Konfigurationen in vielerlei Hinsicht besser abschnitten als eine gewählte Baseline, was die Anwendbarkeit des Ansatzes bestätigt, wurden bestimmte Nachteile entdeckt. Vor allem leidet die Leistung erheblich, wenn ein System mit optimierten Parametern mit einer realistischen Workload-Verteilung statt einer synthetischen konfrontiert wird. Dies betont die Notwendigkeit von genauer Lastenmodellierung und motiviert auch adaptive Ansätze als zukünftige Forschungsaktivitäten.

Abstract

Edge computing addresses bandwidth and latency limitations of the cloud, enabling real-time processing for use cases like the Internet of Things (IoT) and autonomous vehicles. The paradigm shift comes with the need to adapt the orchestration mechanisms of the cloud to the edge. This includes autoscalers, the orchestration components responsible for increasing and decreasing utilized resources according to demand. One of the big challenges holding modern edge-cloud autoscaling solutions back from widespread adoption is the need for static configuration. For example, autoscalers can rely on thresholds that are complex aggregated values and may not be easy to determine beforehand. This thesis explores ways to automatically tune autoscaling parameters ahead of their deployment using a simulation of the target infrastructure. To this end, performance metrics of edge-cloud orchestration are investigated. Eight Key Performance Indicators are chosen as a representative set using a correlation-graph-based approach. Furthermore, a method to aggregate them into a single quality score is presented. An explanation of why the proposed notion of quality only represents one of countless valid definitions is given. Six metaheuristic optimization algorithms are selected and adapted to tackle the task at hand. These are subsequently evaluated against one another using a benchmark scenario of a small smart city edge-cloud deployment. The results indicate that the single-objective approaches Particle Swarm Optimization (PSO) and Artifical Bee Colony (ABC) perform best among the observed candidates. PSO achieved a consistent quality score improvement of around 7.1 % compared to the chosen baseline while keeping computation effort low. ABC achieved an average improvement of the quality score of around 8.6 % and in some cases up to 19.4 %, but required disproportionately more resources to do so. NSGA-II, the investigated multi-objective approach, underperformed. The difficulty in obtaining a well-defined Pareto front suggests that the complexity of the objective space posed a challenge. The robustness of the optimized configurations is analyzed by evaluating their performance under different deployment conditions. While the tuned configurations performed better than a chosen baseline in many regards, showing the presented approach is viable, certain drawbacks were discovered. Most notably, performance suffers significantly when a system using the tuned parameters is faced with a realistic workload distribution across the infrastructure as opposed to a synthetic one. This highlights the need for more accurate load modeling and also motivates more adaptive approaches among possible future research activities.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Over the past decade, cloud computing has emerged as the dominant computing paradigm due to its flexibility and its ability to provide seemingly infinitely scalable resources [17]. However, more recently, application areas have emerged where cloud computing fails to address certain needs of application developers and users. The centralized approach suffers particularly from high network latencies that make it infeasible for domains like autonomous driving and augmented reality. Among other issues are those related to data privacy concerns and bandwidth limitations [3, 43]. To address these demands, edge computing has shifted into the spotlight of researchers. In this setting, requests are processed on smaller compute nodes closer to the client's device, as opposed to powerful servers in data centers. One of the many novel challenges this approach presents is the aspect of user mobility [6]. The nodes servicing the user need to adapt to not break Service Level Agreements (SLA) if the mobile end device changes its location. A useful abstraction in this context is serverless functions in the form of a Function-as-a-Service (FaaS) model. Different approaches have been proposed on how autoscalers, the platform orchestration components responsible for increasing and decreasing the number of deployed functions across the infrastructure, would need to be adapted to fit into a mobility-aware deployment model [1, 32, 63, 66].

Although approaches tackling this issue aim to optimize resource placement at runtime, they usually still require the configuration of static parameters by an expert. An example of this is the approach presented by Raith et al. [66], which introduces the notion of *pressure*. Although pressure was shown to be generally a favorable concept to consider for FaaS-based edge computing platforms, there still remains the need to pick two thresholds of pressure, which determine when the autoscaler performs scaling operations. Hence, it becomes difficult to fully utilize the potential benefits of a novel autoscaling scheme in a

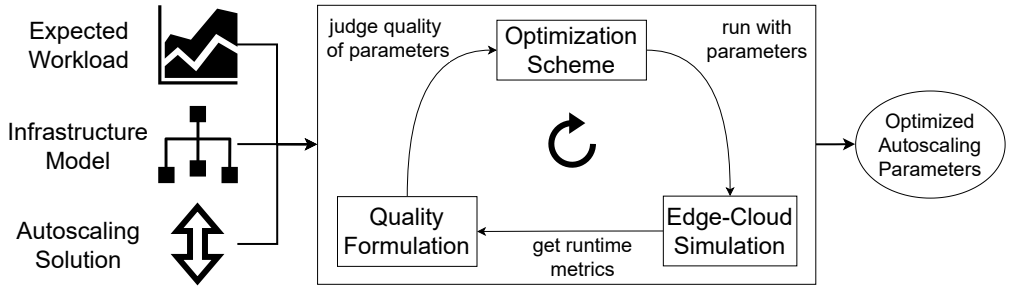

Figure 1.1: Top level view of the approach the thesis aims to introduce.

real deployment. Other suggested platform setups aiming to solve the same issue, such as MCOTM [63] or the one proposed by Huang et al. [32], suffer from the same problem.

A major challenge experts face when setting up such configuration parameters is that the quality of the resulting autoscaling behavior strongly depends on the infrastructure setup, the type of applications that will run on it and the expected request pattern [67].

It can be observed that, despite there being a lot of published work trying to solve the edge computing autoscaling issue, only few approaches are actually deployed in real-world settings and products. Among other issues, Straesser et al. [77] mention the need to configure static parameters as one of the key challenges holding back novel autoscaling solutions from finding practical use.

Hence, a way to automatically optimize these static parameters would represent a valuable contribution to the current body of work. It would solve the problem of needing expert input to set abstract parameters before deployment. This is particularly valuable, as these parameters are often based on assumptions that can only be verified once the solution is deployed. Different novel autoscaling techniques that utilize thresholds or similar concepts could make use of such an approach to solve one of the main issues that plague a sizable portion of published approaches.

1.2 Aim of the Thesis

The aim of the thesis shall therefore be concerned with finding an approach to optimize such autoscaling parameters. The focus is solely put on optimization before the deployment of an edge-cloud platform. This is also the research area where a knowledge gap in the published literature can be identified.

Figure 1.1 shows a high-level view of the kind of system the thesis aims to explore. Given a model of the infrastructure and a set of workload patterns that resemble the expected load on the system, an optimization scheme is used to tune the static parameters of a given edge-cloud autoscaling solution. For the sake of simplicity, the presented work uses the notion of *pressure* for function autoscaling, as proposed by Raith et al. [66], as the

primary example to evaluate against. However, the ultimate aim is to provide insights, that can also be applied to other approaches using different concepts and parameters.

Due to promising contributions in the field of edge computing simulators, a simulation of the target infrastructure is used at the core of the proposed optimization schemes. Specifically, Faas-Sim [65], a simulator that uses couple simulation (co-simulation), serves as an integral part of the optimization process. Co-simulation, where loosely coupled components interact with one another through an event-based model, is a natural fit for the simulation of edge computing infrastructure, which consists of heterogeneous nodes that run and communicate in parallel. It is acknowledged that other such simulators [25, 75, 92] exist and may work just as well or even better for the presented approach. However, their evaluation is not part of the thesis's scope.

The simulator's output is used to extract a set of Key Performance Indicators (KPI) representing relevant aspects of overall orchestration performance. The choice of KPIs is motivated by a literature-based search for the most commonly used quality metrics for edge-cloud orchestration performance and subsequent experimental analysis. These KPIs form the basis for the introduction of a quality function. Formulating such a function allows for the use of single-objective optimization schemes. Additionally, the set of KPIs can be used for multi-objective optimization and also evaluation of subsequent experimental results. The definition of a set of KPIs and the formulation of such a function represent a major contribution of the thesis.

Different optimization approaches are explored. To this end, a promising set of optimization algorithms is extracted from among those that have already been successfully utilized in relevant publications. The scope is limited to metaheuristics, as other forms of optimization schemes, such as exact mathematical models or machine-learning-driven approaches, are either not a good fit for the given problem or would drastically expand the scope beyond reason. Focus is placed on single-objective optimization utilizing the described quality function. To this end, the metaheuristic algorithms Particle Swarm Optimization (PSO), Genetic Algorithm (GA), Differential Evolution (DE), and Cuckoo Search Optimization (CSO) are explored. Additionally, the multi-objective metaheuristic NSGA-II is also considered. This way, assumptions about the applicability of multi-objective approaches to the given setting can be verified. Proof-of-concept implementations of the chosen metaheuristic algorithms are provided and described. Ultimately, the optimization schemes are evaluated against one another on criteria such as quality of results, required computational effort, and convergence behavior to find the best-suited approach among them.

Finally, the applicability of the approach to real-world deployments is evaluated by observing how well a set of optimized parameters performs when there are discrepancies between the optimized and actual settings. These differences include slight differences in infrastructure components, unexpected request patterns, and unanticipated load on the system. The ultimate goal here is to evaluate how robust tuned parameters are compared to those chosen based on basic common sense alone and whether overfitting becomes an issue or not.

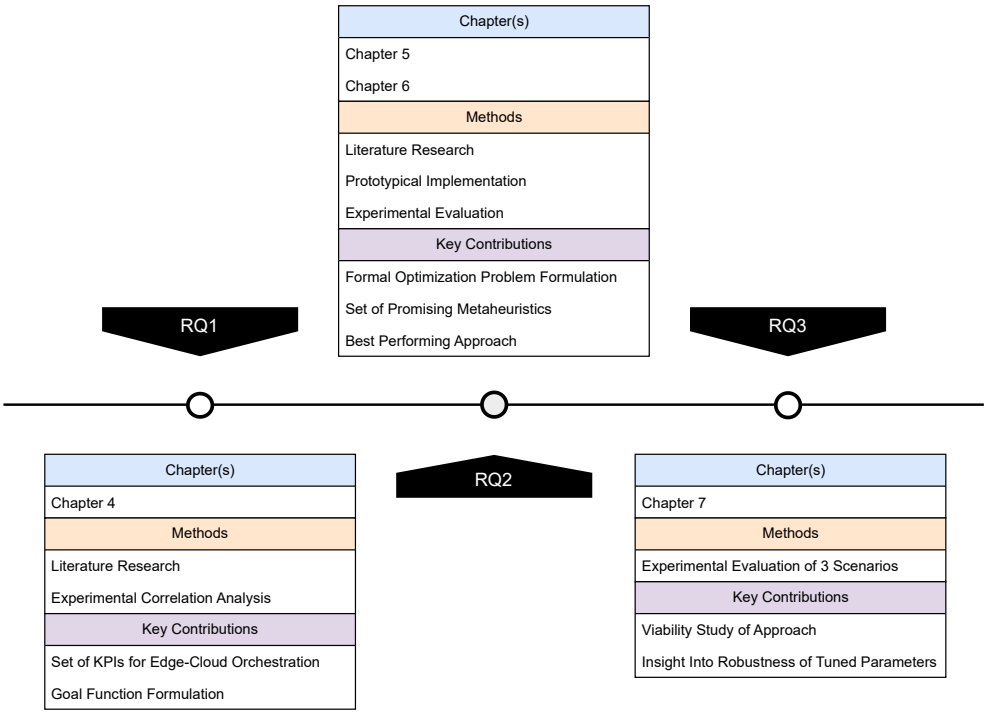

Figure 1.2: High level road map of approach used to answer the research questions.

Hence, the three research questions that the thesis attempts to answer are as follows.

1. How can the quality of an edge-cloud deployment best be estimated based on available runtime metrics to guide an orchestration parameter optimization process?
2. Among promising optimization techniques, which performs best when used to optimize bootstrapping parameters of edge-cloud autoscaling solutions?
3. How robust are the parameters resulting from such a scheme when faced with fluctuations in the infrastructure, request patterns, and load at runtime?

1.3 Approach

Figure 1.2 shows a high-level view of the research activities carried out to answer the three main research questions. It also functions as a reader's guide.

To address the first research question, a large-scale literature review is conducted and a set of relevant metrics is extracted from the available body of work. These metrics are then reduced to a set of eight KPIs for edge-cloud orchestration performance using a method involving a correlation graph. Knowledge gathered throughout the literature

review is then used to distill the eight KPIs down to a possible quality function for autoscaler parameter optimization.

The second research question is explored by first formally defining the optimization problem at hand. The six promising metaheuristics are prototypically implemented, and important details about the decisions made during implementation are discussed. Open hyperparameters of the approaches are systematically selected. Finally, the algorithms are experimentally compared against one another to find the best performing among them concerning the given setting.

Finally, an experimental study is conducted in an attempt to answer research question 3. To this end, three different scenarios are set up to test the robustness of optimized edge-cloud autoscaler parameters: one where the infrastructure is altered, one where the workload is replaced by request patterns observed in the real world, and one where the overall system load is altered. To achieve the second, data from the *Shanghai Telcom Dataset* [24] is used to simulate a workload based on real user data.

1.4 Structure

The remainder of the thesis is organized as follows. Chapter 2 provides background knowledge on the setting and relevant research fields, including basic information on the six chosen metaheuristic algorithms. Chapter 3 outlines existing related work. Chapter 4 details the approach to selecting representative KPIs for system quality and a quality function for single-objective metaheuristics. Chapter 5 formally defines the autoscaling parameter search as an optimization problem, outlines its unique features, and details various choices made during the prototypical implementation of the chosen algorithms. Chapter 6 describes the comparative algorithm evaluation, including experimental setup, hyperparameter choices, and result analysis. Chapter 7 presents three robustness experiments, including setups and result analysis. Finally, Chapter 8 summarizes the thesis, discusses results in relation to the research questions, and suggests future research directions.

CHAPTER 2

Background

This chapter provides important background information related to the conducted research activities. It gives an overview of the concept of edge computing, specifically when utilizing a FaaS model. Focus is put on the orchestration mechanisms a platform implementing these concepts needs to deploy. Of those, particularly autoscaling is highlighted for its relevance to the thesis's goals. A short introduction to metaheuristic optimization is given. The chapter concludes by briefly introducing each of the metaheuristic algorithms that are adapted and evaluated in later chapters.

2.1 FaaS-based Edge Computing

In recent years, cloud computing has emerged as a dominant computing model. It enables convenient ubiquitous access to a set of shared computing and storage resources by making use of pooling and virtualization techniques [54]. A variety of different service models, the prospect of seamless scalability, and the financial aspect of a pay-per-use monetization model are, among others, contributing factors to cloud computing's wide-reaching adoption across the industry.

However, using cloud computing infrastructure, regardless of the service model, does have certain drawbacks. Cloud data centers are usually few in number and, on average, located quite far away from the end user of a cloud-based application. This physical distance incurs latencies, which prevent certain applications from running effectively in the cloud [43]. As an example, one can imagine an assisted driving system that makes critical decisions regarding road safety based on the aggregated sensor measurements of multiple vehicles. Offloading any step of the decision-making process to the cloud would certainly cause vehicles to react too late to successfully avoid safety risks in fast-moving traffic [61]. Other shortcomings of the cloud include data privacy concerns, bandwidth issues, and a lack of location and mobility awareness [3]. These issues particularly plague

Internet of Things (IoT) services, which usually produce large amounts of data, that need to be processed quickly to properly function [53].

As an answer to these issues, edge computing has emerged. The core concept behind edge computing is to move resources closer to the end user's device to overcome the aforementioned shortcomings. Hence, in edge computing, processing happens at the edge of the network, instead of in the cloud, which conceptually sits at the center. In the aforementioned assisted driving example, edge computing could be utilized by introducing roadside base stations that perform the necessary data aggregation and computation, drastically lowering experienced latencies [61]. The paradigm is characterized by its dense geographical distribution, mobility support, location awareness, proximity to the end user, low latency, and device heterogeneity [43]. Edge computing alone can in most cases not fully replace the immense processing resources provided by the cloud. The cloud can, for example, provide backup processing resources in case the nearby edge nodes become over-utilized [66]. Edge computing therefore can be regarded as an extension to the traditional cloud computing paradigm. When considering the full spectrum of possible computation nodes from end devices all the way to cloud data centers, one refers to it as the edge-cloud continuum [42].

Surrounding edge computing, different concepts have been proposed that are worth outlining and clarifying. *Fog Computing* is a term coined by Cisco, which describes a network infrastructure, where networking components, such as routers and switches, are also equipped with computing and storage capabilities [53]. In fog computing, processing can therefore happen everywhere across the edge-cloud continuum. The paradigm has gained considerable traction among recent research publications. Unique concerns, such as hierarchical processing, become relevant for this setting. It can either be seen as an implementation of edge computing principles or as an extension of it beyond the edge of the network [43, 53, 3]. In any case, research focusing on either shares a lot of common ground with the other. On the other hand, *Mobile Edge Computing* (*MEC*) is a proposed edge computing scheme, where Internet Service Providers (ISP) provide computing infrastructure located at the ISP's radio network base stations [61]. It is an implementation of edge computing in the sense that the processing happens at the edge of the mobile network. However, MEC specifically covers only the scenario in which the end device is connected via a radio network, such as 5G. It uniquely enables services to access information about the connectivity of their end devices in the radio network among other context data. *Cloudlet Computing* is a similar concept, where smaller data centers, which are physically distributed, serve end devices instead of a central cloud [72]. The paradigm lacks some of the radio-network-specific service options that MEC offers.

For edge computing, the question of the service model that is provided to users is just as important as for cloud computing. In recent years, the Function as a Service model, also known as *serverless* computing, has emerged as a popular choice for cloud customers [12]. In FaaS, the user provides only stateless functions to the provider and defines events that trigger executions. The deployment of these functions is entirely opaque to the user. Hence, the name *serverless*, as it hides all aspects of infrastructure from the user who

implements a service. FaaS is also attractive for users from a financial point of view, as it offers a pure pay-as-you-go cost model, where they only need to pay for the resources needed while the function actually executes. FaaS is an established paradigm among different cloud providers. Examples of product offerings include AWS Lambda, Azure Functions, and Google Cloud Run functions. This paradigm is also naturally attractive for the scope of edge computing. Desirable aspects include its comfortable pricing model and its fine-grained autoscaling capabilities, which naturally fit together with the resource constraints of edge nodes [6].

Although adapting edge computing to a FaaS-based model brings many benefits and opportunities, it also creates new unique challenges. Among others, the established orchestration mechanisms that serverless cloud platforms use need to be adapted to the edge computing paradigm to properly consider aspects such as node heterogeneity, resource constraints, user proximity, and user mobility [6].

2.2 Serverless Edge Computing Orchestration

In the context of a serverless platform, orchestration refers to the mechanisms that manage the deployment of function replicas across the available infrastructure and the routing of requests to them. Orchestration aims to provide the best possible Quality of Service (QOS) to the end user while keeping the cost to the platform and the service provider low [9]. This naturally includes the efficient use of available resources. The core components of an orchestration framework can be identified as the *Load Balancer*, the *Scheduler*, and the *Autoscaler* [66].

A load balancer is responsible for distributing incoming requests among deployed function replicas with the goal of neither over- nor under-utilizing any replica-hosting nodes [9]. In pure cloud settings, simple approaches, like a round-robin algorithm, can lead to decent results. However, when considering an edge computing setting, where decisions about e.g. cloud offloading need to be made, requirements for load balancers naturally become more complex [39].

The scheduler's responsibility is to place new function replicas among available nodes and to remove them again once they are deemed to be no longer needed [39]. These placement decisions should be made in a way that does not overstrain nodes but also ensures requests to the scheduled replica can be processed in time. When considering an edge setting, this component, in particular, is faced with challenges related to user mobility. The ability of an edge computing scheduler to place resources near where end devices will request access to them is a key performance goal [66]. Furthermore, the inherent heterogeneity of edge devices presents yet another challenge. For example, a scheduler should be able to schedule replicas of CPU-intensive functions primarily to nodes that have stronger processors [39].

Finally, an autoscaler is the orchestration component that decides when to place new function replicas and when to remove running replicas from nodes again [66]. Usually, they

work in intervals. At each interval, the autoscaler evaluates the state of the infrastructure and potentially forwards scale-up or scale-down events to the scheduler. There are two approaches that an autoscaling solution can take: a reactive one, where decisions are only based on the current state of the system, and a proactive one, where some algorithm tries to predict the future state of the system [78]. In a cloud data center, an autoscaling solution needs to only consider a global view of all nodes across the infrastructure. However, when translating to an edge setting, there may be a need to only scale replicas on certain parts of the infrastructure. Using the assisted driving example again, if traffic moves from one road to another, there will be the need to scale up replicas, where traffic is dense, but replicas in the area where traffic thinned out may be removed again. If offloading to the cloud or other parts of the edge infrastructure is considered, the problem becomes even more complex. It is also apparent that the responsibilities of the scheduler and the autoscaler begin to overlap when they are adapted to an edge setting [66].

Various approaches have been proposed to adapt autoscaling mechanisms for deployment on the edge, where setting-specific characteristics such as locality and mobility are considered [1, 32, 63]. One such example is the notion of *pressure*, introduced by Raith et al. [66]. In their proposed platform, they introduce an abstract metric, *pressure*, which gives a sense of where resources are needed compared to where they are currently located. The autoscaler and scheduler subsequently perform their operations in a directed manner based on the pressure values that were calculated. Although many such proposals exist, there is a noticeable gap between the approaches proposed in the literature and autoscaling solutions actually used in the industry [77].

2.3 Metaheuristic Optimization

Mathematically speaking, optimization is considered the process of finding a global minimum or maximum for a given function, referred to as a goal or quality function, subject to a set of constraints [82]. Using methods like Gradient Descent, one can solve simple versions of such problems, where the characteristics of the function are fully understood, without much computational effort. However, it is often the case with real-world problems that the function acts as a black box, where characteristics, such as derivatives or the number of local optima, are not known. More specifically, these problems are a subclass of inverse problems, because, while it is possible to obtain a function value given a set of input parameters, it is impossible to construct an inverse function, where the values of the original function are mapped to their respective inputs [50]. In most cases, such settings make exact mathematical modeling infeasible.

When faced with such a problem, one usually has no other option than to perform a guided search through the input space. Metaheuristic algorithms aim to provide efficient schemes for such guided search approaches that are more or less problem-agnostic. By design, this class of algorithms cannot guarantee that a found solution is, in fact, optimal. However, they are meant to be used in situations where a *good enough* solution suffices and true optimality is not required [55]. Metaheuristics are stochastic by definition,

employing some form of randomness to promote exploration of the search space. Hence, metaheuristic optimization can also be referred to as stochastic search [50].

Metaheuristics are a great choice for solving problems where

- the search space is very large and multi-dimensional,
- exact mathematical modeling is not an option,
- there exist complex constraints on the search space, or
- the optimized function's values change over time [55].

Over the past decades, many different metaheuristic algorithms have been proposed in academic publications [82]. This hints towards the fact that, despite these algorithms being more or less problem-agnostic, no single algorithm has managed to prevail as a definitive choice for any given optimization problem.

No single agreed-upon taxonomy of metaheuristics exists, but based on attempts at definitions found across the literature, the following listing should at least allow for a reasonable categorization of most approaches [45, 46, 82].

- **Non-Nature-Inspired Algorithms**: This group includes metaheuristics that follow simple intuitions, such as Taboo Search and Hillclimbing.
- **Nature-Inspired Algorithms**: These algorithms draw inspiration from nature. The category comprises the majority of proposed approaches and can be further divided.
  - **Physics-Based Algorithms**: Algorithms in this class are based on physical phenomena. A prominent example is Simulated Annealing which is inspired by the way metals behave when cooling down.
  - **Swarm-Based Algorithms**: These algorithms mimic the collective behavior of social insects or animal groups. Examples include Particle Swarm Optimization and Ant Colony Optimization.
  - **Evolutionary Algorithms**: Inspired by Darwin's theory of evolution, these algorithms use mechanisms like selection, crossover, and mutation to evolve a population of solutions. The probably most well-known example in this group is the Genetic Algorithm.

Obviously, this taxonomy does not cover all cases. For example, Cuckoo Search Optimization, a metaheuristic explored further later on, cannot be properly categorized. However, finding further common ground between outlier algorithms becomes difficult, motivating the introduction of a simple *Other* category.

Furthermore, metaheuristics can be split into two groups depending on whether they support multi-objective optimization by design or not. Multi-objective optimization is a setting in which multiple, possibly competing functions are optimized at the same time. This creates the inherent issue that comparing solutions becomes more difficult, as there are now cases where one solution cannot be deemed strictly better than the other. To this end, multi-objective metaheuristics usually follow the concept of Pareto Dominance. Solution A Pareto dominates solution B if it is better in at least one objective and not worse in any other. The set of solutions that are not dominated by any other solution is called the Pareto front and forms the result of multi-objective optimization [55].

It is worth pointing out that, when faced with a setting where multiple objectives can be identified, it is not automatically guaranteed that using a multi-objective metaheuristic will yield the best results. For one, every single-objective metaheuristic can be applied to a multiobjective problem by aggregating the multiple quality functions into a single one. This approach comes with the risk of losing information and, therefore, getting results that may not be located on the Pareto front. However, for simpler scenarios, the obtained solutions often perform well enough, which ultimately is the goal of metaheuristic optimization [50]. Additionally, evolutionary algorithms have emerged as the class of metaheuristics most suitable for multi-objective optimization due to them being inherently population-based [55]. However, this drastically reduces the number of viable approaches available to be explored for concrete problems. Lastly, it has been shown that for a larger number of optimization goals, it becomes increasingly difficult for algorithms to efficiently find a representative Pareto front [50].

One common denominator among most metaheuristics is that they introduce a set of algorithm-specific control parameters, also called hyperparameters [55]. These usually have a significant impact on the algorithm's performance. They commonly control the balance between exploration and exploitation of the search space, but may also steer other aspects of their respective algorithm. Whether a given set of hyperparameters performs well is usually dependent on the characteristics of the problem that is being optimized. Finding a well-performing set of hyperparameters in itself presents an optimization problem and is referred to as hyperparameter optimization (HPO) or meta-optimization [51].

2.4 Explored Metahueristic Algorithms

This section introduces the six metaheuristic optimization algorithms that were chosen for experimental analysis with respect to their suitability for optimizing static edge-cloud autoscaler parameters. Each metaheuristic is briefly introduced and motivated as a choice for the conducted research. The algorithms listed in this section often leave out details for the sake of brevity. The focus is put on the main concepts of the algorithms and important specialties.

The approaches were chosen based on

1. whether they were already successfully used for similar problems across surveyed literature,
2. they have special qualities, making them particularly interesting,
3. or they contributed greatly to the overall variety of investigated schemes.

2.4.1 Particle Swarm Optimization

Particle Swarm Optimization is a well-established metaheuristic for optimizing arbitrary quality functions, first introduced by Kennedy and Eberhart [41] in 1995. It has been extensively studied, applied, and extended since its original proposition [101]. PSO can be categorized as a swarm-based algorithm, inspired by the behavior of flocks of birds. Conceptually, a set of individuals, called particles, iteratively explores the problem space where locations correspond to inputs to the quality function. PSO imposes no restrictions on the quality function's inputs [41, 50]. Therefore, it is a popular choice for optimizing real-valued, high-dimensional problems. However, it can also be used to solve discrete problems by way of search space discretization [44].

The basic flow of the metaheuristic is sketched in Algorithm 2.1. First, the swarm of n particles is initialized. Each particle i saves the following information:

- its current position x_i,
- its personal best found position $pbest_i$, and
- its current velocity v_i, a vector describing heading and speed of the particle in the search space.

Additionally, *gbest*, the overall best position found, is saved and shared across the entire swarm. In each generation, the velocity and position of all particles are updated according to the rules in lines 9 and 10. The control parameters c_1 and c_2 determine how much a particle is drawn to its *pbest* and the global *gbest* respectively. Parameter w is called *inertia* and controls how quickly a particle changes direction. There exist versions that add a separate parameter for scaling a particle's movements. However, most implementations set this value to 1, so it is omitted here [50].

Various extensions to the canonical PSO algorithm have been introduced, such as the Accelerated PSO, which removes velocity and *pbest* from the model to speed up convergence [96], PSO-EA, which combines principles of evolutionary algorithms with PSO [10], and the adaptive PSO, where various hyperparameters are adjusted during runtime of the algorithm [100]. Furthermore, contrary to the classic PSO, which implements information sharing between all individuals, neighborhood-based variations have been proposed. They are based on the idea of limiting information sharing between individuals to a set of

Algorithm 2.1: PSO Algorithm.

Input: quality function $f : \mathbb{R}^d \to \mathbb{R}$
Output: best discovered parameters

initialize n particle positions x_1 to x_n and velocities v_1 to v_n;
for $i \leftarrow 1$ **to** n **do**
  $pbest_i \leftarrow x_i$;
end
initialize gbest;
while ¬ *termination criterion met* **do**
  for $i \leftarrow 1$ **to** n **do**
    Pick random vectors of size d: r_1, r_2 uniformly in range [0;1];
    $v_i \leftarrow w * v_i + c_1 * r_1 * (pbest_i - x_i) + c_2 * r_2 * (gbest - x_i)$;
    $x_i \leftarrow x_i + v_i$;
    if $f(x_i) < f(pbest_i)$ **then**
      $pbest_i \leftarrow x_i$;
      if $f(x_i) < f(gbest)$ **then**
        $gbest \leftarrow x_i$;
      end
    end
  end
end
return *gbest*

neighbors. Various topologies for such approaches have been studied [68]. Although many of these variations make compelling promises regarding convergence speed and algorithmic performance, no variations of the standard PSO were chosen for further investigation to keep the scope of this work reasonable. However, there is clear merit in experimenting with PSO variations, leading to a recommendation for potential future work.

PSO was chosen to be among the analyzed optimization schemes due to its widespread successful application in relevant settings [2, 71, 60], its natural suitability for real-valued, multi-dimensional optimization problems and its straightforward parallelizability. However, an often criticized aspect of the classic PSO is that it, in general, requires many quality function evaluations and that convergence happens rather slowly, which has prompted many adaptations of the original algorithm.

2.4.2 Genetic Algorithm

The Genetic Algorithm is a metaheuristic inspired by mechanisms of biological evolution, natural selection, and genetics based on the fundamental work of John Holland in the 1970s. Hence, it falls under the category of evolutionary algorithms. The GA assumes that a solution to an optimization problem can be represented as a combination of separate

Algorithm 2.2: GA Algorithm.

Input: quality function $f : \mathbb{R}^d \to \mathbb{R}$
Output: best discovered parameters

```
1  initialize n individuals x_1 to x_n;
2  initialize best;
3  while ¬ termination criterion met do
4  |   for i ← 1 to n do
5  |   |   fitness_i ← f(x_i);
6  |   end
7  |   select parents;
8  |   perform crossover;
9  |   perform mutation;
10 |   replace individuals;
11 |   update best;
12 end
13 return best
```

building blocks, referred to as chromosomes. These chromosomes should follow the schema theory, which in this context states that small atomic parts of a solution contribute independently to the overall quality of the larger solution [70]. These chromosomes are subjected to operations representing metaphors for biological evolution, such as mutation, recombination, crossover and selection. Although the GA is typically used to solve discrete problems, it is also possible to use it for real-valued function optimization [50].

The basic steps of the canonical GA are outlined in Algorithm 2.2. First, a population of individuals is initialized. An individual is essentially a set of chromosomes that encode a possible set of inputs for the quality function. Individuals can be initialized either randomly or according to a problem-specific strategy. In either case, diversity in the initial population is encouraged to emphasize exploration in the early phases of the algorithm. In the main loop of the algorithm, each individual's fitness, which refers to the result of the function to optimize, is evaluated. Importantly, the GA assumes a maximization problem. Subsequently, the population is subjected to three nature-inspired operations.

During *Selection*, individuals are selected for mating. There are different established selection strategies for deciding which parents will produce offspring. Popular options are as follows.

- *Roulette Wheel Selection* (RWS), where an individual's likelihood of being selected is proportional to its fitness.
- *Rank Selection*, where the individuals are first ranked according to their fitness, and then a number of top individuals are selected for mating.

- *Tournament Selection*, where individuals metaphorically face off against each other in a tournament between random opponents.
- *Stochastic Unversial Sampling* (SUS), which functions similarly to roulette wheel selection but guarantees that the fittest individuals are included in the selection.

During *Crossover*, the genetic material of the mating parents is recombined to create new offspring that have traits of all parents. There exists a multitude of crossover strategies, such as single-point, two-point, or uniform crossover. However, the best strategy usually depends on the problem instance and its genetic representation. The frequency with which crossover is performed is driven by a control parameter.

The *Mutation* step is important for introducing diversity into a population. Here, chromosomes are randomly altered. Similarly to crossover, there are multiple ways to do this, such as swapping chromosomes or performing arithmetic operations on them. Once again, the right choice depends on the problem and the genetic representation chosen for it. There is yet another control parameter which determines the rate at which mutation occurs.

Finally, at the end of an iteration, individuals are replaced by the offspring. In the classic GA, a child generation fully replaces the parent generation. However, there exist variations of the algorithm that allow promising parents to remain in multiple generations. When using *elitism*, for example, a defined number of parents with high fitness are automatically transferred to the next generation [15, 50]. This ensures that the currently best solutions always contribute to the gene pool. However, this may discourage exploration. Alternatively, in the *steady-state* GA only a small number of children is created in each iteration and replaces individuals of a larger population. This way, fitter individuals contribute even more to the gene pool, which again may negatively impact exploration [93, 50].

The GA was chosen for investigation, mainly for its widespread use among recent publications, despite originally being proposed around half a century ago [14, 11, 4]. Additionally, the GA's strengths are that it is highly customizable and parallelization is straightforward. However, there may be drawbacks to using it in the given scenario. Namely, the lack of a straightforward bit vector encoding of the search space, the large number of hyperparameters requiring tuning that the GA offers, and the fact that the GA naturally requires a large number of quality function evaluations may be problematic.

2.4.3 Artificial Bee Colony

Like PSO, Artificial Bee Colony Optimization is a metaheuristic that takes inspiration from swarm behavior in nature, with the difference that ABC is inspired by bees instead of birds. The metaheuristic was originally proposed by Karaboga and Basturk [40] and can be used to optimize arbitrary real-valued functions but it can also be applied to discrete problems similarly to PSO [44]. In ABC, individuals are divided into three groups: workers, onlookers, and scouts. Workers are responsible for refining solutions

Algorithm 2.3: ABC Algorithm.

Input: quality function $f : \mathbb{R}^d \to \mathbb{R}$
Output: best discovered parameters

initialize w food sources;
initialize *best*;
for $i \leftarrow 1$ **to** w **do**
  $limit_i \leftarrow l$;
end
while $\neg$ *termination criterion met* **do**
  for $i \leftarrow 1$ **to** w **do**
    generate and evaluate new candidate food source for w_i;
  end
  for $i \leftarrow 1$ **to** o **do**
    assign onlooker o_i to food source of worker w_j with probability $\frac{f(w_j)}{\sum_{n=1}^{w} f(w_n)}$;
    generate and evaluate new candidate food source for o_i;
  end
  replace assigned worker food sources if improved;
  for $i \leftarrow 1$ **to** w **do**
    if *assigned food source of* w_i *was not improved* **then**
      $limit_i \leftarrow limit_i - 1$;
      if $limit_i = 0$ **then**
        scout for new food source and assign w_i to it;
        $limit_i \leftarrow l$;
      end
    end
  end
  update *best*;
end
return *best*

that have already been discovered. Onlookers are responsible for driving the optimization towards more promising solution spaces. Scouts are tasked with finding new potential but still unexplored solutions. Hence, the distribution of individuals assigned to each group fundamentally determines the balance between exploration and exploitation.

In ABC, a candidate solution is called a food source, and the corresponding value of the quality function is referred to as nectar or fitness. The more nectar, the better a food source is considered. Hence, ABC assumes a maximization setting. The algorithm is divided into three main phases, one for each group of individuals, which are repeated in each iteration. It is outlined in Algorithm 2.3. First, for each worker, a food source is randomly initialized. In each generation, every worker generates a new candidate food

source and measures its nectar. They then share this information with the onlooker bees. Onlookers probabilistically choose a food source based on its associated fitness. For each onlooker, a new candidate solution is generated in the same manner. The original authors propose generating the new candidates based on the equation

$$v_{ij} = x_{ij} + \phi_{ij} * (x_{ij} - x_{kj}), \tag{2.1}$$

where v is the new candidate solution, x is the original food source, ϕ is a vector of random values in the range of [-1, 1], k is the index of a randomly chosen food source different from x and j is a randomly chosen index of the solution vector. Hence, each time a new neighbor is generated, only one parameter will be altered. Because new solutions are entirely based on existing food sources, as the algorithm converges on smaller areas, the variations of new candidate solutions also decrease in magnitude. Should a refined solution be found that improves the one assigned to the worker, it then becomes assigned to the worker for the next iteration. Afterwards, there is a scouting phase. Each worker keeps track of a *limit*. The initial value l represents a control parameter. In each iteration, if no candidate solution for the assigned food source yielded an improvement, the limit is reduced by 1. If the limit hits 0, the food source is considered exhausted and is replaced by a random food source discovered by a scout. Subsequently, the worker's limit is reset again. In the original proposal, the authors limited the maximum number of scout bees to 1, meaning that each exhausted food source is simply regenerated entirely randomly [40].

It is worth pointing out that ABC has two separate parameters controlling the overall population size, which sets it apart from other population-based optimization schemes presented. The authors of the original paper recommend using the same number of workers as onlookers [40]. However, using different ratios may also be beneficial.

ABC was included among the analyzed metaheuristics because of its unique metaphorical model and its ability to easily be tuned toward more exploration or more exploitation of the search space. Additionally, it has fewer hyperparameters to tune overall than other comparable approaches and there are examples of it being used effectively in relevant fields [99].

2.4.4 Differential Evolution

Similarly to the GA, Differential Evolution is a population-based evolutionary algorithm which, in contrast to the GA, follows a more simplified approach. It was originally envisioned by Storn and Price [76] and refined in a series of publications. DE is most often used to optimize real-valued problem spaces, but can also be adapted to solve discrete problems [44]. Once again, the algorithm works with a population of individuals, where an individual's position represents a set of input parameters to the problem's quality function. DE models individuals as vectors and relies on vector arithmetic for its evolution-inspired mechanisms. Although the algorithm has the advantage of reduced complexity, it has been shown to struggle with multi-modal optimization problems [34].

Algorithm 2.4: DE Algorithm.

Input: quality function $f : \mathbb{R}^d \to \mathbb{R}$
Output: best discovered parameters

initialize n individuals x_1 to x_n;
initialize *best*;
while ¬ *termination criterion met* **do**
  for $i \leftarrow 1$ **to** n **do**
    pick 3 random individuals a, b, c distinct from x_i and each other;
    pick index $ri \in [1, d]$ randomly;
    initialize new position $x'_i \leftarrow x_i$;
    for $j \leftarrow 1$ **to** d **do**
      pick number $r \in [0, 1]$ randomly;
      if $j = ri \vee r < cp$ **then**
        $x'_{ij} \leftarrow a_j + dw * (b_j - c_j)$;
      end
    end
    if $f(x'_i) \leq f(x_i)$ **then**
      $x_i \leftarrow x'_i$;
    end
  end
  update *best*;
end
return *best*

DE is outlined in Algorithm 2.4. First, the individuals are initialized. In each iteration, each individual's position is modified using three other random but distinct individuals by combining their vector components arithmetically. A control parameter cp, called crossover probability, determines how likely it is for each vector component of the original position to be modified. By introducing the random index ri, it is guaranteed that at least one position is always modified. A further control parameter dw, called differential weight, is used to control the magnitude of the mutation. Once a new set of positions has been generated, their respective costs are evaluated. If a new individual is better than the one on which it was based, it replaces it in the population. The listed vector operations follow those originally proposed by Storn and Price [76]. However, there exist other valid ways to define crossover operations on vector level [48].

DE was included in the set of analyzed metaheuristics because it is based on the same foundation as the GA but follows a simpler model that also requires tuning fewer hyperparameters. The hope is that these simplifications will lead to decent performance while reducing quality function evaluations. Furthermore, it has recently been successfully used to optimize problems in relevant domains [34]. However, DE may struggle with the potentially multi-modal topology of the search space.

Algorithm 2.5: CSO Algorithm.

Input: quality function $f : \mathbb{R}^d \to \mathbb{R}$
Output: best discovered parameters

initialize n nests;
initialize best;
while ¬ *termination criterion met* **do**
  pick nest n_c from nests randomly;
  get cuckoo position x_c by Lévy flight from n_c;
  pick nest n_i randomly;
  if $f(x_c) > f(n_i)$ **then**
    replace n_i with x_c;
  end
  evaluate all nests;
  sort nests according to cost;
  for $k \leftarrow 1$ **to** $ceil(p_a * n)$ **do**
    pick the k-th worst nest n_a to abandon;
    generate new nest n_r;
    replace n_a with n_r;
  end
  update *best*;
end
return *best*

2.4.5 Cuckoo Search Optimization

The Cuckoo Search Optimization Algorithm is another nature-inspired metaheuristic that models the unique behavior of cuckoo birds to hide their eggs among nests of other bird species. It was originally introduced by Yang and Suash Deb [97]. The core of the algorithm revolves around the observation that birds, among other flying species, chart courses, which can be modeled by steps taken from a Lévy distribution: A so-called Lévy flight [69]. A Lévy distribution is a probability distribution of positive values characterized by a long tail. Lévy flights therefore include many small changes in position interspersed by larger hops, which provides a more efficient search pattern for heuristic optimization. Therefore, CSO is essentially an optimized random search scheme. CSO can be used to solve arbitrary real-valued optimization problems of any dimensionality, but common discretization techniques can be used to adapt it for discrete problems as well [44].

The algorithmic flow is outlined in Algorithm 2.5. In CSO, a candidate solution is called a nest. First, a set of nests is initialized. In each iteration, a random nest is picked by the metaphorical cuckoo, to start a Lévy flight from. The cuckoo's position is calculated by adding a step to each parameter of the original nest's position. The step size is taken from a Lévy distribution. Subsequently, to properly scale the step, it is multiplied by

a parameter α controlling the step size. Because the Lévy distribution only produces positive values, the sign of the step is flipped with a 50 % probability. Subsequently, the quality of the cuckoo's solution is compared with that of a random nest. If a better solution was found, the nest is replaced by the cuckoo's position. The algorithm, as originally proposed, assumes a maximization problem. However, it can be converted to a minimization problem simply by flipping the comparison in line 7. Finally, in each iteration, the worst-performing nests are abandoned. The fraction of replaced nests is controlled by the control parameter p_a. Each abandoned nest is replaced by a newly generated one.

Walton et al. [89] created a modified version of the CSO. They introduce some constraint handling approaches, an alternative way to generate new nests and a mechanism for information exchange between nests. Should CSO prove to be a promising metaheuristic for the given setting, further exploration of this approach is motivated as potential future work. Furthermore, Ghodrati and Lotfi [21] introduced a hybrid approach that combines CSO with PSO, which is also not explored further.

CSO was chosen to be among the investigated approaches because, as opposed to the other metaheuristics presented, it is essentially an advanced guided random search scheme. Additionally, CSO has the important advantage of having very few hyperparameters to tune and those that can be tuned have been shown by the original authors to not significantly impact convergence behavior [97]. Finally, there are reports of successful applications of CSO in relevant fields among the surveyed literature [29, 49, 57].

2.4.6 NSGA-II

The Non-dominated Sorting Genetic Algorithm II is a multi-objective evolutionary algorithm, originally proposed by Deb et al. [16]. Essentially, it extends the GA introduced in Subsection 2.4.2 to a multi-objective version by altering the parent selection scheme to use non-dominated sorting and crowding to determine which individuals are chosen for reproduction. Additionally, NSGA-II is fundamentally elitist. As is the case with all multi-objective optimization algorithms, NSGA-II does not return a single optimal set of parameters for the given function, but instead creates a Pareto front of multiple Pareto-optimal solutions. The GA's fundamental characteristics, that it is more suited for discrete problems and that it relies on the schema theorem, are also true for NSGA-II [50, 16].

The scheme conceptually follows the same steps outlined in Algorithm 2.2. However, parents are not selected based purely on fitness, as there now exist multiple goals. Instead, non-dominated sorting is used to rank all individuals according to their Pareto rank. Subsequently, the individuals within the ranks are sorted again according to their crowding distance. Individuals who are more isolated with respect to their position in the solution space, spanned by the objectives, are preferred to promote diversity in the population. Depending on the approach, either the top individuals according to this sorting are selected for reproduction or a tournament selection, where Pareto rank and crowding

distance determine the winner, is used. To achieve inherent elitism, the algorithm places the selection step at the start of an iteration and always considers the children and their parent generation for selection. Furthermore, NSGA-II uses some algorithmic optimizations to allow non-dominated sorting in $O(MN^2)$ time complexity, where M is the number of objectives and N is the number of individuals in the population [16].

NSGA-II was included, to also add a multi-objective optimization scheme to the list of analyzed metaheuristics. No other multi-objective scheme was included, as NSGA-II has often been shown to be one of the best-performing approaches in this class aside from specific settings [50, 23]. Furthermore, it has been used to successfully optimize static parameters and runtime decisions in edge-cloud-related settings [67, 80]. Finally, it is worth noting that multi-objective metaheuristics are inherently more computationally intensive. Especially when there are a larger number of objectives – usually four is named as a sensible limit – it can take many iterations to arrive at a representative Pareto front [50].

CHAPTER 3

Related Work

This chapter covers existing publications related to the main research areas of the thesis. It gives an overview of state-of-the-art work covering attempts at defining edge-cloud orchestration quality, examples of optimization techniques being used to solve problems across the edge-cloud continuum, and existing comparative studies concerning metaheuristics in related fields.

3.1 Orchestration Quality

Across the body of existing literature, there are multiple publications that attempt to compile a set of KPIs that represent the quality of edge-cloud orchestration solutions. In the course of their survey on the state of the art of application placement solutions in fog computing, Nayeri, Ghafarian, and Javadi [56] compile a list of 30 distinct performance metrics organized into five groups observed in existing publications.

An alternative list of KPIs is presented by Luo et al. [52] in the course of their survey on resource scheduling in edge computing. Their list contains six course-grained metrics and lacks a taxonomic structure.

Goudarzi, Palaniswami, and Buyya [22] focus their survey on edge IoT application scheduling on creating a taxonomy of possible approaches. One dimension they introduce is that of *Optimization Characteristics*, which also includes six groups of target metrics. However, they list *Other* as a distinct group and do not exhaustively list all KPIs that fall into this category.

In the course of establishing a framework for recommending edge-cloud orchestration architectures, Pouresmaeil [62] introduces a set of 10 KPIs organized into the four measurement levels of *System*, *Network*, *Application*, and *Orchestration Level*. This taxonomy is based on the preceding work of Böhm and Wirtz [9], who introduce a quantitative evaluation approach for edge orchestration strategies.

Aslanpour, Gill, and Toosi [5] present 65 distinct target metrics for edge-cloud computing performance. They organize them on the basis of whether the metrics relate to the *cloud*, *edge*, *fog*, *IoT* or a *common* domain.

The heterogeneity of published lists of KPIs already hints towards the fact that no ultimate single source of truth for what KPIs are truly relevant for edge-cloud orchestration performance can be identified. However, the existing body of work serves as a good foundation for the research activities outlined in Chapter 4.

3.2 Parameter Optimization in Edge-Cloud Settings

Optimizing autoscaling, among other orchestration mechanisms of cloud and edge platforms, is a highly relevant topic among recently published research. Relevant work dealing with different aspects of the topic is presented next, grouped by covered topics of increasing similarity to the thesis's approach.

3.2.1 Parameter Optimization in Cloud Computing

In an early publication, Al-Haidari, Sqalli, and Salah [26] study the effects of the static parameters *CPU threshold* and *scaling size* on the performance of cloud clusters. They propose a simple empirical method for finding optimal values for these parameters for a specific simulated application deployment.

Taherizadeh and Grobelnik [78] argue that tuning the static parameters of the Kubernetes autoscaler can have tremendous effects on the performance of a cloud-hosted application. They identify three key parameters, that require special consideration, namely *control loop time interval*, the maximum *number of instances* that can be shut down per scaling interval and a special constant α proposed by them, which determines how quickly the cluster's configuration may change.

A large body of published work shifts the optimization problem to the runtime, by not attempting to optimize static parameters of infrastructure setups but instead attempting to find the optimal placement of processes at every scheduling interval. HUNTER [88] and its extension HunterPlus [36] use a machine learning approach based on a *Gated Graph Convolution Network* to find the optimal task placement in cloud and fog settings. These publications primarily focus on increasing energy efficiency in cloud data centers while keeping SLA violations to a minimum.

Toka et al. [81] propose using three competing machine learning algorithms for the optimization of the placement problem in cloud data centers. They criticize the static and unintuitive parameters of the Kubernetes autoscaler and reduce the needed user input down to a single parameter, which controls the balance between minimizing either cost or SLA violations.

In the context of edge computing, a popular approach to scheduler optimization is to use reinforcement learning (RL) to continuously improve the resource placement strategy

of the system in dynamic environments. Contributions by Xu, Chen, and Ren [95], Chen et al. [13], and more recently Wang et al. [90] show successful applications of this technique to optimize placement strategies in an edge computing setting regarding different goals.

A different solution, specifically designed with mobility in mind, is proposed by Abdullaev et al. [1]. They use a deep learning approach and tune the parameters of their model using a metaheuristic based on seagull flocking behavior. The approach focuses specifically on optimizing offloading strategies for IoT clients in the edge-cloud continuum.

Huang et al. [32] propose a FaaS-based edge platform and develop a function placement and migration optimization strategy by formulating the problem into integer linear programming and using receding horizon control to solve it.

3.2.2 Using Simulations for Optimization

Infrastructure simulations are often used as tools to evaluate novel contributions to the fields of cloud and edge computing. However, there are still examples of simulators being successfully utilized as a component of optimization schemes themselves.

An early approach in this category is proposed by Hiroshima and Komoda [28]. They simulate a cloud deployment in the form of a highly reduced mathematical model and use it as a basis for optimizing their autoscaling approach in a cloud setting.

EdgeTuner [27, 91] addresses one of the core issues of RL-based approaches. Namely, that the bootstrapping of the algorithm takes a significant amount of time, which is detrimental to the quality of the scheduling decisions right after deployment. In EdgeTuner, the K8sSim [92] is used to pre-train the RL-based optimizer using historical trace data.

In their introduction to the Skippy scheduler for serverless edge computing, Rausch, Rashed, and Dustdar [67] also propose the use of a simulation to optimize weights used in their scheduler's goal functions. The authors formulate a multi-objective optimization problem with four goals and use NSGA-II to solve it. They show a significant performance boost of the optimized parameters across three different infrastructure settings but do not provide a deeper analysis of other possible optimization schemes.

3.2.3 Co-Simulation based optimization

Due to the heterogeneous and highly parallel nature of cloud and edge computing infrastructures, it is a naturally fitting use case for the paradigm of co-simulation. The idea of using co-simulation directly as part of an optimization strategy is not novel. GOBI [87], GOSH [85], SimTune [86] and CLIP [84] all utilize a co-simulator representing a digital twin of the target infrastructure as a core component of their scheme for optimizing autoscaling decisions. GOBI, GOSH and SimTune explicitly address the issue of resource scheduling in edge-to-cloud infrastructures, while CILP is focused on the provisioning of virtual machines in a pure cloud computing scenario. They all use the COSCO [87] simulator to achieve this. However, these proposed schemes do not utilize

the simulator as a quality function of some optimization algorithm, but instead train a neural network to mimic the simulator's output. This approach helps speed up the actual evaluation of placement decisions that is once again taking place at runtime and also enables gradient-based optimization strategies to be used.

None of these preceding publications directly solve the issue the thesis is aiming to tackle. The papers concerned with optimizing static parameters are cloud-specific and offer limited value for the relevant setting, which focuses on FaaS and edge computing. The trend for edge computing settings regarding this topic seems to lean heavily in the direction of performing optimizations during runtime. Although there is clear value for the goal of the thesis to be found in these approaches, the main focus, which is the optimization of static parameters, cannot be fully solved by the presented research alone.

3.3 Comparison of Metaheuristics in Edge-cloud Settings

Among published work, there are publications in which the authors evaluate different metaheuristic optimization approaches when used in a setting related to that of the thesis. These shall be outlined next.

Guerrero, Lera, and Juiz [23] compare three multi-objective metaheuristics, NSGA-II, a weighted GA and MOEA/D, regarding their suitability to solve the service placement problem in a fog setting. They conclude that results obtained from the NSGA-II were better quality-wise, but MOAE/D performed well while requiring less execution time overall.

Nguyen et al. [58] introduce a novel optimization algorithm called TCaS to solve the task scheduling problem for IoT in a fog setting. In their evaluation, they compare their algorithm with the two established metaheuristics Bee Life Algorithm and Modified Particle Swarm Optimization. They find that their approach performs better in the given setting.

Zafar et al. [98] compare the suitability of Particle Swarm Optimization and a metaheuristic called Bat Algorithm to solve resource allocation problems in a fog computing scenario. They conclude the Bat Algorithm to be the better of the two.

Hussain et al. [33] introduce a novel multi-objective optimization heuristic specific to resource allocation in vehicular fog networks. They compare their approach with the established metaheuristics NSGA-II and SAMPSO, concluding that their approach is superior.

All of the listed publications reduce the set of evaluated metaheuristics to at most three. Most of them also use comparative analysis to prove the superior quality of a novel solution introduced in the same paper. The contribution of this thesis differs in the sense that a set of diverse and already established metaheuristics is evaluated regarding their suitability for optimizing static autoscaler parameters. However, the existing literature provides adequate guidelines for the proper evaluation of optimization schemes guiding the approach used in Chapter 6.

CHAPTER 4

Quantifying Edge-Cloud Orchestration Quality

This chapter is concerned with exploring possibilities for gauging the quality of an edge-cloud deployment, focusing on the performance of its orchestration mechanisms. The aim is to ultimately use this notion of quality to drive an optimization process to tune static autoscaler parameters.

Firstly, the endeavor is motivated in the greater scope of the thesis, and the challenges of the approach are outlined. A literature review is presented, which was conducted with the goal of outlining different metrics that have been used in the past to represent the quality of various aspects of edge-cloud deployments. Based on these results, a subset of metrics relevant to the outlined goal is presented, including the ways that other researchers have normalized, aggregated, and combined them. An experimental setup is described, which is used to gather data on the relationships of said metrics. Finally, the results are analyzed, combined with the knowledge gathered through literature research, and then used to reduce the presented metrics to a representative set of KPIs. The metrics in that set are then normalized and combined to formulate a quality function that will be used for parameter optimization going forward.

4.1 Motivation and Challenges

As described in Section 1.2, the ultimate goal of the thesis is to use metaheuristic approaches, such as evolutionary algorithms and swarm intelligence algorithms, for the optimization of static deployment parameters. All of these approaches share a common core. They are concerned with finding a good, but not necessarily optimal, solution to a minimization problem. Hence, it is necessary to be able to map a certain configuration of parameters, subject to tuning, to a set of numerical values, when using multi-objective

approaches, or a single value in the case of single-objective optimization [55]. In both cases, lower values should indicate a favorable set of parameters.

This value shall reflect the overall quality of the system, as it can be assumed that the configuration of the orchestration components has a considerable impact on the overall performance of the platform. For example, if an autoscaler is configured to perform more downscaling operations than necessary, the users of the platform will experience worse response times, due to increased queuing times resulting from nodes becoming congested.

However, formulating such a notion of quality is not straightforward due to various challenges that are inherent to the given paradigm.

1. There are a plethora of metrics available to be aggregated into such a quality function. These are, however, quite heterogeneous and no general consensus exists on which ones to prefer over others that may measure similar aspects of the deployment's runtime behavior.

2. Quality itself is an abstract concept that needs to be defined on a per-case basis and relates to the fulfillment of stakeholder interests, of which there are multiple in the given setting.

3. Stakeholders in this scenario have competing goals. For example, a user wants to experience the lowest possible response times, while a platform operator will want to keep operating costs to a minimum. These different viewpoints need to be considered and it is challenging to combine them into a single notion quality [62].

It is obvious that no single formulation of quality can be considered perfect. Depending on the viewpoint taken, certain aspects of platform performance will be more or less important. Despite these challenges, the rest of this chapter will be concerned with attempting to arrive at a common notion of quality, with the goal of it being as universally applicable as possible. Efforts will be based on existing approaches among the published literature and an experimental exploration of the relationships of a set of selected metrics. However, it is acknowledged that any formulation of quality will ultimately be flawed in some respects. Hence, any subsequently analyzed optimization schemes will keep the choice of the notion of quality flexible and refrain from deeply interweaving it in their design.

4.2 Literature Review

4.2.1 Metrics

This section is intended as a general overview and rough taxonomy of metrics that have been used as the basis for the formulation of a quality measurement in edge, fog, cloud or combined settings among current research. The metrics must not necessarily have been used for the purpose of heuristic optimization. For example, RL-based autoscalers require

Publication	Time	QOS	Resource	Cost	Reliability
Raith et al. [66]	RTT		CPU		
	Lat		Replicas		
Rausch, Rashed, and Dustdar [67]	FET		Cloud Traffic	AWS model	
			Proc. Loc.		
Han et al. [27]	PT				
Tuli, Casale, and Jennings [84]	RT	SLO-rate		real deployment	
	WT			Energy	
Tuli et al. [87]	RT	SLO-rate	Fairness	Energy	
	PT				
	MT				
	SchT				
	WT				
Tuli, Casale, and Jennings [86]	RT	SLO-rate		undisclosed model	
	SchT			Energy	
	WT				
Huang, Lan, and Xu [30]	RT				
Huang, Liang, and Ali [31]	RT				MTTF
					MTTR
Ghobaei-Arani and Shahidinejad [20]	SD	SLO-cons	Proc. Loc.	Energy	
Guerrero, Lera, and Juiz [23]	Lat		CPU		
			RAM		
			Service Spread		
Tuli et al. [88]	RT	SLO-rate	CPU	Energy	
	SchT		RAM	Temperature	
	WT		Fairness	Azure model	
	MT				
Iftikhar et al. [36]	SchT	SLO-rate		Energy	
	WT			Temperature	
Wang et al. [90]	SD				
Abdullaev et al. [1]	SD			Energy	

Table 4.1: Metrics used for judging edge-cloud system quality in reviewed literature.

the formulation of a reward function. Although RL has to take place during runtime and the reward function serves primarily for steering a machine learning process, which differs from the described setting, there is clear value in examining the utilized metrics. Another setting, where a notion of orchestration quality is necessary, is the evaluation of novel solutions in the field where the same line of reasoning applies. Therefore, the surveyed literature is not limited to publications only concerned with autoscaling solutions, but also includes publications dealing with other platform orchestration topics such as loadbalancing and scheduling.

Tables 4.1 and 4.2 list the surveyed literature and the metrics utilized in the publications. A basic taxonomy of metrics is presented next. Similar collections are given by the publications listed in Section 3.1. However, these listings are neither exhaustive nor free of redundancies, nor set their focus on the setting the thesis is concerned with.

Publication	Time	QOS	Resource	Cost	Reliability
Lera, Guerrero, and Juiz [47]	RTT	SLO-rate	CPU		Avail.
			RAM		
			Hard Drive		
Xia et al. [94]	RTT				
	Lat				
	SchT				
Nazir et al. [57]	RT			own model	
	PT				
Hussein and Mousa [35]	RT				
Canali and Lancellotti [11]	Lat				
	PT				
Akintoye and Bagula [4]	Lat			own model	
	PT			Energy	
Tang et al. [79]	RTT		CPU	own model	
			RAM	Energy	
			Hard Drive		
Skarlat et al. [74]	RT	SLO-cons	Proc. Loc.	own model	
	Lat				
Bhatia, Sood, and Kaur [7]	RTT			Energy	
Zafar et al. [98]	RT				
Gazori, Rahbari, and Nickray [19]	RTT	SLO-ot		Azure model	
	RT	SLO-rate		Energy	
	WT				
	SD				
Tran et al. [83]	RT	SLO-cons	Proc. Loc.	real deployment	
				Energy	
Chouat, Abbassi, and Graiet [14]	PT				MTTR
Hong et al. [29]	SchT	SLO-ot	Proc. Loc.	own model	
	WT			Energy	
	RT				
Liu et al. [49]	SD	SLO-ot	Proc. Loc.	own model	
	RT			Energy	

Table 4.2: Metrics used for judging edge-cloud system quality in reviewed literature continued.

Time-based Metrics

The most common category of metrics that was used throughout the gathered literature is concerned with different notions of time. Nayeri, Ghafarian, and Javadi [56] identify a total of 17 different target metrics relating to time, the most relevant of which are described below.

Round Trip Time (RTT) is the broadest notion of time, measuring the wall clock time between the moment a user sends a request and the moment the fully processed response arrives back at the client. This metric most accurately captures the system's responsiveness from a user's point of view. However, of the surveyed papers, none that use an online optimization approach utilized RTT. This is likely because, from a platform perspective, the final delay between sending the response and the user receiving it usually cannot be measured. RTT was a common metric used across discovered approaches that use offline optimization and for the evaluation of novel solutions [66, 47, 94, 79, 7, 19].

Response Time (RT) spans the wall clock time from the moment a request is received at the service provider's infrastructure to the moment a response is sent. It is equal to the RTT minus the network delay between the client device and the gateway of the service provider in both directions. This metric is by far the most utilized in the reviewed literature [84, 87, 30, 31, 88, 47, 57, 35, 74, 98, 19, 83, 29, 49]. It is often favored by approaches that perform runtime optimizations, as it is easily measurable. Furthermore, this metric deliberately ignores any network latency caused by communication across channels that are not directly influenced by the service provider, which can be favorable for certain viewpoints. It is important to note that the distinction between RTT and RT is not always made this clearly, with publications sometimes referring to RTT as RT.

Latency (Lat) refers to the wall clock time a request or response spends in transit while traveling through networking components. Hence, it is the time that a request is neither actively processed by a compute node nor waiting in a queue for processing. One can differentiate between internal latency, pertaining to network components within a controlled system, and external latency, pertaining to communication links across the Internet. This metric is a key indicator for systems that aim to optimize resource placement to minimize network delays [66, 23, 94, 4, 11, 74].

Processing Time (PT) only considers the wall clock time it takes for a request to be processed. Delays through network communications, scheduling, queuing, or other overheads are deliberately ignored. In the context of FaaS, this metric is also known as *Function Execution Time* (FET) [67]. The main goal, when inspecting this metric, is to get a notion of how well compute resources are used [67, 87, 57, 11, 4, 14]. A high average or peak PT should indicate congestion among some of the processing nodes, leading to slower computation. Hence, there is also an argument to be made to list this metric as a resource-focused metric instead.

Service Delay (SD) describes the total delay caused by platform overhead. It encompasses the total time that a request spends in the system not actively being processed, including internal latencies and task queuing times. This metric gives an overall view of the

severity of managerial platform overhead. Service delay is particularly sensitive to bad orchestration configuration, as poor autoscaling can lead to extensive queuing times and lots of cold starts, where function invocations have to wait for a new replica to be scheduled before executing, adding a considerable amount of delay [90, 20, 1, 19, 49].

Other metrics observed in the surveyed body of work, which are noteworthy, are *Scheduling Time* (SchT), the time it takes for the scheduler to find a node to place a replica on [87, 86, 88, 36, 94, 29], *Waiting Time* (WT), the time scheduled replicas spend in a queue before being ready to execute requests [87, 86, 88, 84, 36, 19, 29], and *Migration Time* (MT) in a setting where tasks may be migrated between different nodes while running [87, 88]. MT in particular is only relevant for settings that support long-running tasks, for example, ML training workflows. While these metrics contribute to the overall responsiveness of the system, the autoscaler's behavior only has a limited effect on them.

Furthermore, there are proposed schemes, like *EdgeTuner* [27], that differentiate between jobs and tasks, where one job may include multiple tasks. In such a scenario, times can be measured on both job- or task-level. The FaaS-based model assumed in this work does not make this distinction.

QOS-based Metrics

While minimizing timing metrics, like RTT, may be the primary objective from a user's perspective, these metrics do not paint the whole picture when taking the service provider's viewpoint into account. Usually cloud and edge services are provided under certain predetermined Service Level Agreements, which consist of Service Level Objectives (SLO), determining how long a user can expect a response to take at most. Such SLOs are also referred to as deadlines. Depending on the type of agreement, violations of such an SLO could incur financial losses for the service provider and should be avoided. However, as long as response times are below the agreed-upon SLOs, the service provider is at liberty to optimize other aspects that keep costs low on their end. Hence, SLO violations form the basis for a popular category of metrics.

Some approaches surveyed modeled their optimization schemes in a way that includes deadlines as hard constraints (SLO-cons), which may not be violated [74, 20, 83]. Other approaches accept that SLO violations may happen and use them as part of an aggregated metric that is to be minimized. One option is to try to minimize the rate at which violations occur (SLO-rate) [84, 87, 86, 88, 36, 47, 19]. Alternatively, one can aim to minimize the total overtime of finished tasks (SLO-ot) [19, 29, 49].

Resource-usage-based Metrics

The main resources that are of interest when discussing edge-cloud deployment performance are CPU cores, RAM, and network bandwidth [66, 23, 88, 47, 79]. Sometimes hard drive storage is also considered [47, 79]. However, persistent storage is, in contrast to the other mentioned resources, not critical for every type of service. It is generally in the service provider's interest to make efficient use of these resources, as expending more

than necessary can incur higher costs or degrade the QOS the platform can deliver to clients. Hence, CPU, RAM, and network utilization are key metrics in this category.

However, the raw aggregated utilization does not paint a complete picture. For example, it is also of interest that the system does not deploy unnecessarily many replicas [66] and that resources are fairly consumed across all nodes of the infrastructure. For instance, in their evaluation of COSCO [87] and HUNTER [88], the authors calculate the *Jair Fairness Index* as a target metric to satisfy this goal. Additionally, when looking at an edge cloud setting, it is in the best interest to use edge resources first, before resorting to cloud resources. Hence, some publications use the ratio with which requests are processed on the edge vs. on in the cloud (Proc. Loc.) [67, 20, 74, 83, 14, 29, 49] and bytes transferred between the edge and the cloud [67] as key performance metrics.

Another way to model adequate versus unwanted resource usage is to define certain utilization thresholds similar to SLOs and measure violation rates [79]. A metric uniquely proposed by Guerrero, Lera, and Juiz [23] is *Service Spread* which captures how much replicas are spread out over a given set of infrastructure nodes.

Cost-based Metrics

When looking at a provider's perspective, the main objective in most cases is to keep operating costs low. Cost is an easy-to-measure metric in running systems [84, 83]. However, it is not trivial to predict ahead of time. Estimating costs always requires a cost model that transforms low-level metrics into equivalent monetary value.

Looking at the perspective of a service provider using rented infrastructure, the model can be based on the cost of real edge-cloud platforms [67, 86, 88, 19]. For example, when using AWS [73] as a reference model, the cost drivers are the number of requests received, the total processing time, and the memory allocated. Models based on other providers exist. Alternatively, some researchers conceive own cost models [49, 29, 74, 79, 4, 57].

For a platform provider, who owns the running hardware, the main contributor to cost is *Energy Usage*, which in itself is often used as a performance metric. Furthermore, some researchers differentiate between the energy used to run the hardware and the energy used to cool it. Hence, *Temperature* is also sometimes observed as a key metric [84, 87, 86, 20, 88, 36, 47, 4, 79, 7, 19, 83, 29, 49].

Research focusing on certain aspects of edge-cloud systems may break cost down into smaller contributing factors. Nayeri, Ghafarian, and Javadi [56] identify *Resource Usage Cost*, *Deployment Cost*, *Execution Cost*, *Migration Cost*, *VM Cost* and *Data Transfer Cost* as different contributing factors to the total cost of a system.

Reliability-based Metrics

Discovered publications that focus on how reliable a platform deployment is used established reliability metrics like *Mean Time to Fail* (MTTF), *Mean Time to Repair* (MTTR) and *Service Availability* (Avail.) [31, 47, 14].

4.2.2 Aggregation

To grasp a valid picture of the performance of an edge-cloud deployment, low-level metrics, like the ones presented above, need to be collected over a representative interval of the runtime. Hence, it becomes necessary to aggregate observed fine-grained metrics, such as those falling into the request-oriented timing category. Moreover, for metrics that are decoupled from single requests, for example, resource usage, it becomes necessary to implement reasonable points of measurement. In any case, the question of how to aggregate a collection of runtime metrics is not trivial.

Some of the observed publications opt to use an average over all measured values [67, 87, 30, 31, 88, 36, 90, 57, 35]. Others opt to sum up the measurements [27, 20, 23, 88, 36, 71, 94, 57, 11, 79, 98, 19, 83, 29, 49]. However, these approaches do not consider the impact of outliers. Although this may be intentional, particularly when looking at RT or RTT, the timings experienced in the worst-case scenarios, also sometimes referred to as *tail latency*, are not irrelevant. Hence, approaches such as the one presented by Raith et al. [66] work with percentiles of timing metrics, for example, using the 95th percentile of RTTs. This leads to a more pronounced impact of high tail latencies on the overall quality score. Calculating averages over only the top n % of measurements is also a viable option. An alternative solution is presented by Tuli, Casale, and Jennings [85] in their publication on GOSH. They incorporated the variances of metrics into the aggregation to calculate a *Value at Risk* metric for a 95 % confidence interval.

4.2.3 Normalization

As already outlined, different metrics reflect different aspects of the needs of different stakeholders. Therefore, when reducing multiple metrics to a single value representing overall system quality it brings with it the issue of normalization because the metrics presented in Subsection 4.2.1 have different units associated with them. For example, RTT is usually measured in milliseconds and CPU utilization in percent, two units that cannot be naturally combined in a meaningful way without normalizing first.

An approach for this, often observed in the surveyed literature, is min-max normalization between 0 and 1 [84, 87, 20, 71, 19]. A great benefit of this approach is that the combination with other metrics, which are already represented as percentages, such as the SLO violation rate, follows naturally. However, for some metrics, particularly time-based ones, this approach is plagued by two issues:

1. A minimum or maximum value is not always given before measuring, and
2. values may not be evenly distributed between the maximum and the minimum.

For example, the maximum experienced RTT during runtime is unpredictable. Additionally, though 0 might seem like a reasonable minimum, most RTT measurements will be significantly higher, skewing values toward the upper end and hindering comparability.

Furthermore, since in an optimization setting we need to be able to compare isolated runs, the minimum and maximum values may not change between them, therefore eliminating the option to simply take them from the sampled values.

Therefore, not many publications combine timing-based metrics with non-timing-based metrics [87, 32, 57, 79, 19, 29, 74]. The online optimization approaches among them have the benefit that they do not require comparability between sampling periods and can simply use minimum and maximum from the currently observed timeframe [87, 19]. To combat this problem in the scope of offline optimization, one can work with predefined SLAs, assuming the agreed-upon SLO as the maximum value [74]. However, violating timings would then be normalized to values above 1. Yet another option is to use a separate isolated runtime interval to sample values from. However, this also does not solve the issue of potential values above 1, because higher maxima may be observed in later intervals.

Often it is simply accepted that some metrics will not strictly adhere to the predefined interval [32, 57, 79, 74, 29]. While in theory, this does hurt comparability, in practice it does not prevent optimization schemes from working correctly, albeit possibly less efficiently. For the minimum, 0 was chosen for most metrics that did not have a natural lower bound. Offline approaches preferred predefined SLOs as the upper bound, while online approaches usually kept track of the highest observed value and adapted their normalization strategy.

As a final side note, the challenges presented above are major drawbacks of using single-objective optimization schemes for the presented purpose. Approaches that instead use non-dominated sorting, such as NSGA-II [16], do not suffer from these problems, as they do not require normalized objective values [67, 23, 49].

4.2.4 Combination

Among the surveyed literature, the overwhelming majority of approaches that combined multiple metrics did so using a weighted sum [27, 84, 87, 20, 88, 36, 71, 79, 74, 29]. Formally, given n metrics $m_1, m_2, ..., m_n \epsilon \mathbb{R}$ and n constants $\alpha_1, \alpha_2, ..., \alpha_n \epsilon [0, 1]$ we can combine all n metrics into a total score as follows.

$$\text{score} = \sum_{i=1}^{n} \alpha_n * m_n \tag{4.1}$$

The constant α-values act as weights with which certain metrics can be prioritized over others. Optionally, each term can also include a constant value acting as an offset. However, not many instances of this technique being used were found in the surveyed literature.

4.3 Towards a Quality Function

This section aims to rationalize a preferred subset of key metrics representing system quality and introduce a formulation of a quality function for optimizing edge-cloud autoscaler parameters. First, the metrics presented in Subsection 4.2.1 are reduced, partially transformed, and slightly expanded. These metrics are then observed over a variety of experimental runs of an edge-cloud simulator. The results of these experiments are subsequently analyzed to find correlations between the different metrics. Ultimately, the final set of KPIs is rationalized, and a formulation of a quality function that will be used for the rest of the thesis is present and motivated based on the experiment's results.

4.3.1 Selected Metrics for Evaluation

Table 4.3 gives an overview of all metrics that were chosen as the basis for the conducted correlation experiments. Different ways of measuring or aggregating the same type of metric are organized into groups. They were all chosen such that a lower value indicates a more desirable result. The main criteria for choosing these metrics were

1. their prevalence in the reviewed literature,
2. the possibility to obtain them from a FaaS-Sim [65] run, and
3. the likelihood of them being influenced by the utilized autoscaler parameters.

RTT was chosen as the main temporal metric to investigate, as it most closely translates to quality experienced by end users, and an accurate reading of it is available in FaaS-Sim. In addition to a mean over all requests, averages for the top 10 %, 5 % and 1 % of RTTs are included. This is to give insight into tail latencies and allow analyzing how much they differ from aggregations over all requests. Furthermore, an SLO was defined and the percentage of requests that missed the deadline was measured as an SLO violation rate. This way of measuring SLO-based metrics was preferred because it allows for comparison between settings, where the total number of received requests differs. Counting the number of violating requests or summing the total overtime are equally valid ways to measure SLO violations but lack this quality.

Furthermore, the same metrics based on PT are observed. Since the simulation operates in a FaaS-based setting, the domain-specific term FET will be used going forward. This metric was included as it should provide insight into user-experienced time behavior and efficiency of resource usage of the platform. Notably, an SLO-based metric is also collected for FETs. This is something not observed in the surveyed literature. However, it was still done, as the SLO violation rate provides a convenient normalized version of the timing metric, and this way full parity between RTT- and FET-based metrics is achieved.

To investigate orchestration-level resource usage, metrics related to processing requests in a different zone and specifically in the cloud instead of on the edge are included. For both,

Group	Metric	Abbreviation	Unit
RTT	mean over all requests	RTT	ms
	mean over slowest 10 %	RTT@90	ms
	mean over slowest 5 %	RTT@95	ms
	mean over slowest 1 %	RTT@99	ms
	SLO violation rate	RTT-SLO-rate	%
FET	mean over all requests	FET	ms
	mean over slowest 10 %	FET@90	ms
	mean over slowest 5 %	FET@95	ms
	mean over slowest 1 %	FET@99	ms
	SLO violation rate	FET-SLO-rate	%
Processing Location	cloud-processing rate	CvE-rate	%
	out-of-zone processing rate	ZC-rate	%
Service Delay	mean service delay rate	SD-rate	%
Cost	AWS Lambda@Edge-based model	cost	$
CPU Utilization	mean average utilization	CPU-avg	%
	max average utilization	CPU-max	%
	variance of utilization	CPU-var	N/A
	mean SLO violation rate	CPU-SLO-avg	%
	max SLO violation rate	CPU-SLO-max	%
	variance of SLO violation rate	CPU-SLO-var	N/A
RAM Utilization	mean average utilization	RAM-avg	%
	max average utilization	RAM-max	%
	variance of utilization	RAM-var	N/A
	mean SLO violation rate	RAM-SLO-avg	%
	max SLO violation rate	RAM-SLO-max	%
	variance of SLO violation rate	RAM-SLO-var	N/A
Replicas	mean nr. of deployed replicas	rep	#
	replica budget rate	rep-rate	%

Table 4.3: Metrics selected for correlation experiments.

the ratio of requests that fall into the respective categories compared to total requests is tracked. These metrics are referred to as *zone crossing rate* and *cloud vs. edge rate*.

To measure total platform overhead, a metric is introduced that represents the average percentage of time a request spends not actively being processed: *service delay rate*. It specifically measures the time between the moment the request is received by the platform and the processing starts, and again between the moment when processing finishes and the response is sent by the system. This was done to get a proper reading of relevant overhead such as queuing times and cold start delays.

To estimate the cost, a cost model based on AWS Lambda@Edge [73] is introduced. According to the documentation, the contributing factors to the runtime costs of an edge-cloud service running on the AWS platform are the number of requests made and the total calculation time used on the edge and the cloud, respectively. Processing time is measured in Gigabyte-Seconds incorporating both CPU and memory requirements of the deployed functions in discrete increments.

System-level resource utilization metrics focus on CPU and RAM. Hard drive space is not particularly relevant in the context of serverless edge computing because persistent storage is usually realized by delegating to storage-specific services. The nodes' allocated resources are measured at the time of allocation, as opposed to the actual usage. Since CPU and RAM usage is measured per node, aggregation of measurements becomes nontrivial. To cover all interesting cases, the average resource utilization throughout the whole simulation time is calculated for each node. Then the mean and maximum of this aggregate over all running nodes are observed as metrics. In addition, the variance is also observed to get a picture of how evenly the load is distributed across the system. A higher variance in average resource utilization should point towards a skewed distribution of workload. Additionally, an alternative approach is also implemented, which is based on the idea that resource utilization should not exceed a certain target threshold for each node. Since this viewpoint is similar to SLOs, with the difference that they are relevant for the service provider instead of the user, these metrics are also referred to as SLOs. An SLO violation is measured every time a function replica is allocated to a node that is already above the given utilization target. The same statistical aggregates as for the raw utilization are collected for the rates of violations, which refers to the total fraction of simulation time that the respective node is violating the SLO.

Lastly, the number of deployed replicas is analyzed. Also, similarly to CPU and RAM utilization, a target threshold, referred to as a budget, is defined, which should not be exceeded. The fraction of simulation time, where more replicas than the budged allows for are running, is observed and is referred to as *replica budget rate*.

4.3.2 Experimental Setup

This section outlines the experimental setup that was created to gather information on metric correlations. Experiments are conducted using the FaaS-Sim [65] simulator. FaaS-Sim reports metrics of simulation runs using *Pandas* [59] dataframes. The metrics observed

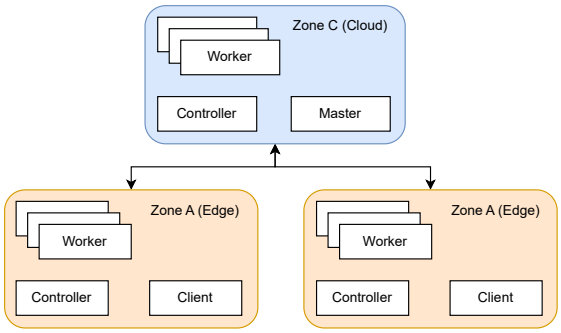

Figure 4.1: Infrastructure used for metric correlation experiments.

in the experiments were implemented according to the specifications in Subsection 4.3.1 by computing the described aggregations over data returned in said dataframes.

Infrastructure

All simulation runs use the same infrastructure setup, which is visualized in Figure 4.1. It is a model of a decentralized edge-cloud deployment consisting of three zones. Zones A and B model edge computing clusters, from which user requests can originate. Zone C models a cloud computing center that originally holds the initial function replicas and also offers computation resources. Because the infrastructure is decentralized, each zone has its own local controller responsible for local orchestration tasks. On top of that, a master node, responsible for global orchestration tasks, is located in zone C. Computation is performed by worker nodes, of which there are two types: strong and weak ones. Strong nodes are modeled to have 8 CPU cores at a clock speed of 2.1 GHz and 16384 MB of available RAM. Weak nodes are modeled to have 2 CPU cores at the same clock speed and 3072 MB of available RAM. Edge zone A consists of one strong node and three weak nodes, edge zone B consists of one strong node and two weak nodes, and cloud zone C consists of three strong nodes. This infrastructure model was introduced by Pouresmaeil [62] in their work, focusing on recommending certain architecture patterns for different edge-cloud use cases. It was chosen because the infrastructure's capacity to handle certain request loads, assuming decently well-chosen orchestration parameters, was already known.

Function

To keep the setup simple, only one function exists that users can invoke on the edge-cloud system. The resource requirements and runtime of the simulated function are based on measurements of a Python function that approximates pi that was run on hardware mimicking the simulated one. It requires a fixed amount of 262144 KB of memory and half a CPU core, which are allocated when scheduling a replica to a worker node. This function is designed to be primarily compute-intensive and requires disproportionately more compute resources than memory resources.

Performance Goals and Parameters

An RTT-based SLO of 600 ms is assumed for the deployed function. This deadline is chosen rather tight but achievable to ensure enough SLO violations happen to allow meaningful analysis of the QOS-based metrics. Analogously, the FET-based SLO is set to 400 ms.

For resource SLOs, 50 % usage was chosen for RAM-, CPU- and replica-based metrics. This is lower than what one would set in an actual deployment scenario, but this was once again chosen to ensure that SLO violations happened at a high enough frequency.

The cost model that is used is based on the pricing information Amazon listed for AWS Lambda@Edge[73] on 23.08.2024. Therefore, the values chosen are 0.6 $ per million requests processed, 0.00005001 $ per GBs of edge-processed requests and 0.0000166667 $ per GBs of cloud-processed requests.

Autoscaling Setup

Kubernetes is used as the orchestration framework. For the autoscaling solution, the already introduced pressure-based approach [66] was chosen, as it will also be used for parameter optimization. Therefore, each zone is assigned a minimum and a maximum pressure threshold per function, p_{min} and p_{max}, which control when up and down scaling operations should be performed.

To simulate a plethora of possible setups, a total of 200 random but plausible scenarios were generated and simulated five times each. Each simulation runs for 3 minutes of simulation time with an autoscaling interval of 5 seconds. For each scenario, the pressure thresholds of each zone and the pattern of arriving requests were varied.

The pressure thresholds were chosen at random with the following constraints:

1. $p_{min} \geq 0.1$,
2. $p_{max} \leq 0.9$,
3. $p_{min} < p_{max}$, and
4. increments are made in steps of 0.05.

This should result in reasonable, but possibly suboptimal, sets of pressure thresholds.

Workload

For request patterns modeling client behavior, six separate request profiles were created. A profile is essentially a list of inter-arrival times of requests. To synthesize these profiles, the *Request Generator* project, first introduced by Raith et al. [64], which is publicly

available on Github[1], was used. Each profile aims to simulate a plausible pattern of arriving requests. Each has a duration of 2 minutes of simulation time. This was chosen such that after the last requests are sent, the system has 1 minute left to process all of them, before the run ends, ideally allowing even very suboptimally configured deployments to process all requests.

The synthesized profiles are visualized in Figure 4.2. Each profile is generated according to a mathematical model and then altered using random fluctuations to arrive at a more realistic arrival pattern. The light blue line shows the exact requests per second (RPS) that are sent each second of the simulation. The dark blue line represents a 10-second rolling window over incoming traffic and gives a better picture of the long-term request behavior. The profiles have been constructed with the following intentions and parameters:

- **Constant**: This profile aims to simulate a high but constant load on the system. For the first 15 seconds, requests gradually ramp up to an average rate of 15 RPS which is then maintained for the remainder of the simulation.
- **Sine**: This profile models the RPS as a sine wave, to simulate a fluctuating load on the system. The sine wave reaches a peak RPS of 30 and follows a constant period of 30 seconds. Additionally, the first 15 seconds are dampened to allow for a more gradual ramp-up in requests.
- **Random Walk**: This profile aims to model a realistic workload by using a random walk starting at 0 RPS and increasing or decreasing each second randomly following a normal distribution with a standard deviation of 1.
- **Spikes**: This profile models a low consistent load on the system with sharp intermixed spikes. The spikes are modeled using the peak of a sine wave, reaching a maximum of 50 RPS and lasting 10 seconds each.
- **Early Load**: This profile models the same behavior as the Constant profile but cuts off after 1 minute.
- **Late Load**: This profile models the same behavior as the Constant profile but only starts to ramp up after 1 minute.

By assigning a request profile to each edge zone, combined workloads were created, with which the simulation is then run. There are two types of workloads, *symmetric* ones that use the same request profile for each zone, and *asymmetric* workloads that use different request profiles for the zones. The asymmetric workloads are visualized in Figure 4.3, where each color represents load originating from a different zone.

[1] https://github.com/edgerun/request-generator

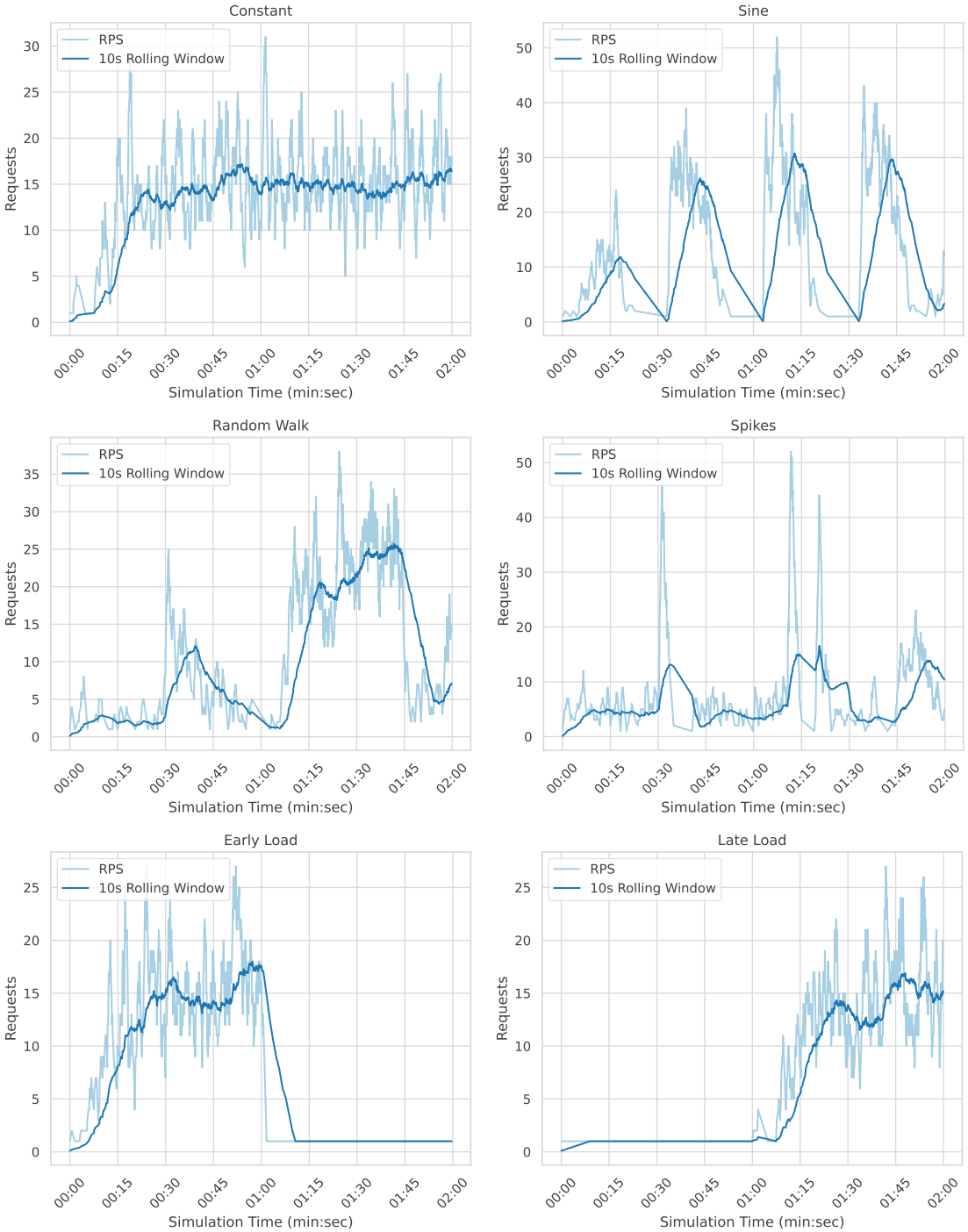

Figure 4.2: Request profiles used in metric correlation experiments.

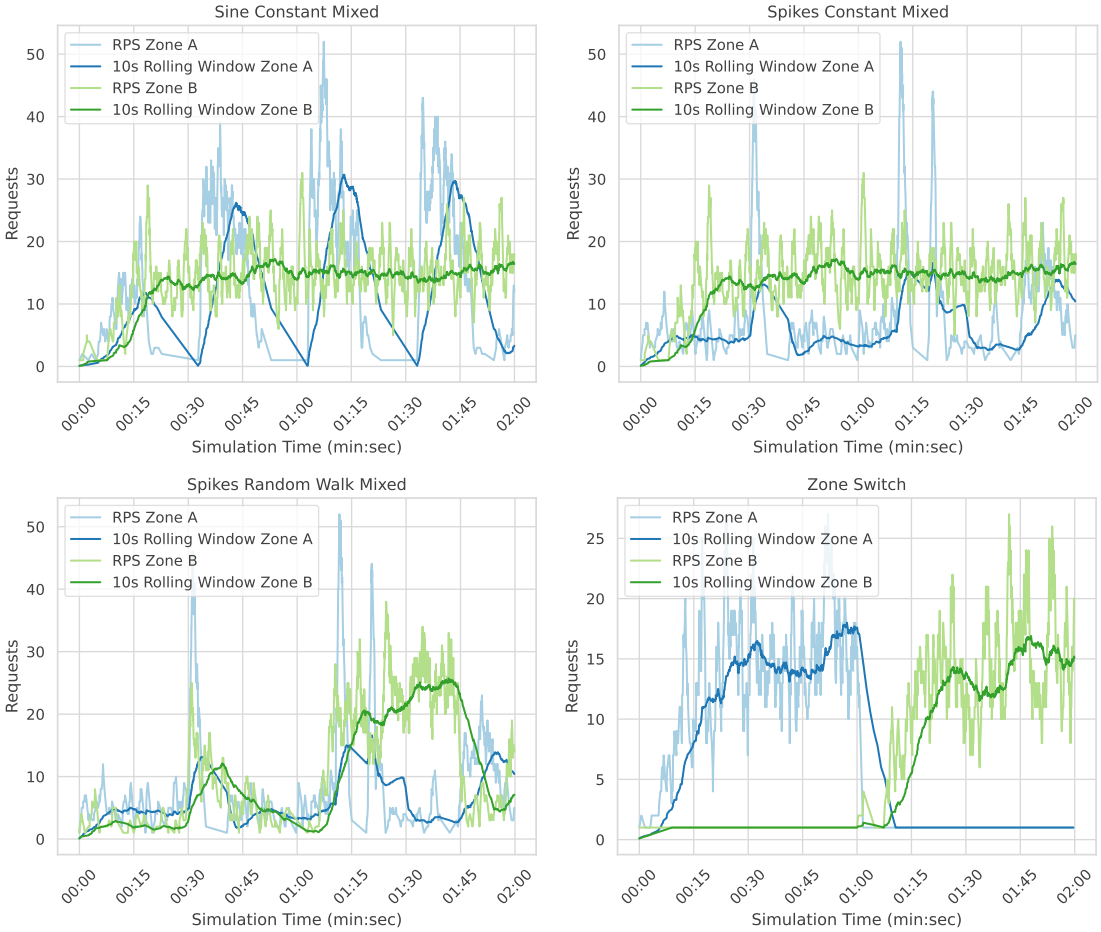

Figure 4.3: Asymmetric workloads consisting of mixed request profiles used in metric correlation experiments.

The following workloads were used:

- **Symmetric Workloads**
  - **Constant**: A workload using the Constant profile for both zones. It aims to simulate a high, but constant load coming from both edge zones.
  - **Random Walk**: A workload using the Random Walk profile for both zones. It aims to simulate a realistic workload on the system with equal load coming from both zones.
  - **Sine**: A workload using the Sine profile for both zones. It aims to simulate fluctuations in the overall load of the system
  - **Spikes**: A workload using the Spikes profile for both zones. It aims to simulate unexpected system-wide spikes in load.

- **Asymmetric Workloads**
  - **Sine Constant Mixed**: A workload that combines the Constant profile and the Sine profile. It aims to simulate a situation where the system is under constant high load from one zone, while the other alternates between a low and a high load. This is meant to test the system's ability to scale up and down resources in one zone while the other has a predictable, but high load.
  - **Spikes Constant Mixed**: A workload that combines the Constant profile and the Spikes profile. It aims to simulate a situation, where both zones produce constant load with one being more demanding than the other, while the lower RPS zone experiences random spikes in requests. This is meant to simulate situations where the systems may have allocated too few resources to one zone and might struggle to compensate for request spikes.
  - **Spikes Random Walk Mixed**: A workload that combines the Random Walk profile and the Spikes profile. It aims to simulate a similar situation as Spikes Constant Mixed but with a more realistic pattern.
  - **Zone Switch**: A workload that combines the Early Load profile and the Late Load profile. It aims to model an extreme scenario in which the system experiences a high load coming from one zone at a time that switches zones after half of the simulated time. This is meant to test whether the system can properly adapt in situations where a large portion of load-generating clients move from one zone to another.

4.3.3 Results

Since all of the metrics analyzed during the experiments are continuous and real-valued, the Pearson Correlation Coefficient is calculated to uncover linear correlations between them. The results of this are visualized as a heatmap in Figure 4.4, where red cells indicate a positive correlation, white cells indicate a correlation close to 0, and blue cells indicate anticorrelation. The actual coefficients are given in each cell rounded to one decimal position.

Macro-Scale Analysis

At a glance, it becomes clear that the analyzed metrics can be divided into two main classes depending on their correlation with the other class.

- CPU-, RAM- and replica-based metrics and
- other metrics including those pertaining to RTT, FET, SLO, SD, processing location and cost.

Within the group, one can generally observe a high positive correlation. Between the groups, there is either a negative, a weak positive, or no correlation to be observed.

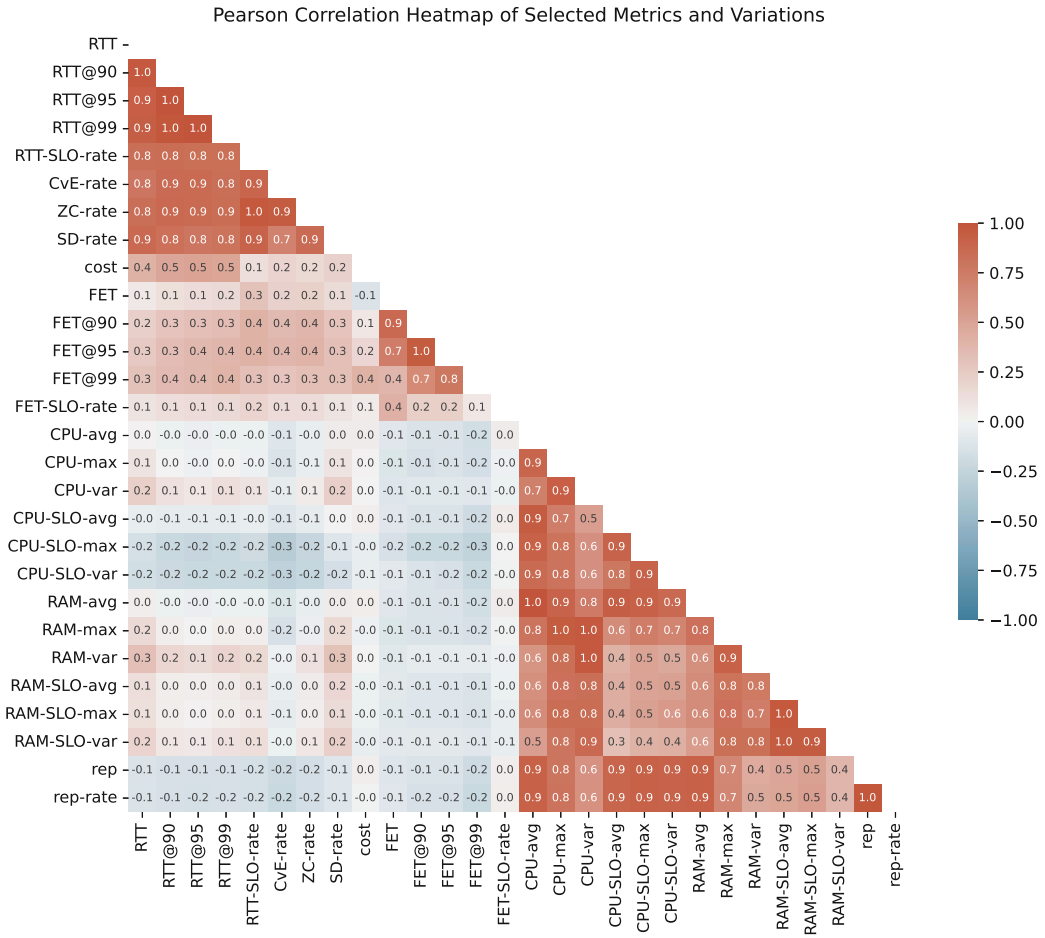

Figure 4.4: Results of the metric correlation experiments as a heatmap.

This indicates that RAM and CPU utilization and the number of deployed replicas are the main aspects whose performance indicators decrease as the user-facing performance indicators, such as RTT, increase.

Because AWS's cost model only considers where the requests are processed and how long it takes to do so, the estimated cost is also indirectly proportional to the node-level resource usage.

A surprising result is that FET-based and CPU-centric metrics are slightly anticorrelated. For example, FET and CPU-avg have a correlation coefficient of -0.0989 and FET@99 and CPU-SLO-max even go as low as -0.2595. The assumption that FET can be effectively used as a metric for measuring how efficiently resources are utilized is therefore challenged. Possibly, there is another contributing factor to the total FET, that is more prevalent than the CPU utilization on a given node.

	RTT	RTT@90	RTT@95	RTT@99
RTT@90	+0.9652			
RTT@95	+0.9365	+0.9930		
RTT@99	+0.9396	+0.9874	+0.9933	
RTT-SLO-rate	+0.8393	+0.8449	+0.8285	+0.8294

Table 4.4: Correlations between RTT-based metrics.

	FET	FET@90	FET@95	FET@99
FET@90	+0.8695			
FET@95	+0.7491	+0.9584		
FET@99	+0.3840	+0.6682	+0.7796	
FET-SLO-rate	+0.4168	+0.2300	+0.2325	+0.0819

Table 4.5: Correlations between FET-based metrics.

Correlations Within Groups

The metrics within the same group are expected to be heavily correlated. However, the degree of this correlation can be a key indicator of whether one variation can be exchanged for another without loss of information.

Looking at the RTT-based metrics, one can observe that all of the point-wise correlations listed in Table 4.4 are, as expected, very high. However, there is a noticeable difference between the percentiles. Although RTT and RTT@90 have a very high correlation of +0.9652, the correlation drops for the higher percentiles with +0.9365 and +0.9396 for RTT@95 and RTT@99, respectively. This indicates that there is more value in observing the 95th and 99th percentile in addition to the overall RTT than when using the 90th percentile. Furthermore, RTT@95 and RTT@99 are also highly correlated with a correlation coefficient of +0.9933. Therefore, there is little value in observing both metrics and picking one has no particular benefit over the other. Looking at the SLO rate, the correlations with the other RTT metrics are considerably lower, albeit still rather high, with all correlations sitting at over 0.8. This indicates that while looking at the SLO rate of RTTs preserves most information conveyed by that metric, there is still some that is lost to the normalization process and these types of metrics are not perfectly interchangeable. Lastly, it would appear that the slowest 10 % of RTTs contribute the most to the SLO violation rate with a correlation coefficient of +0.8449 between RTT@90 and RTT-SLO-rate.

A very different picture is painted by the correlations between FET-based metrics listed in Table 4.5. Here, correlations are much lower than one would expect with the extreme case of the correlation between FET@99 and FET-SLO-rate of +0.0819, indicating that the two metrics are almost perfectly independent. A possible explanation for that is that the FET metric simply did not fluctuate much during experiments with different settings. Across the executed simulation runs, FET shows a standard deviation of 0.006 ms, while for comparison, RTT has a standard deviation of 0.73 ms. One can deduce that either

	CPU-avg	CPU-max	CPU-var	CPU-SLO-avg	CPU-SLO-max
CPU-max	+0.8885				
CPU-var	+0.7326	+0.9363			
CPU-SLO-avg	+0.9484	+0.7379	+0.5335		
CPU-SLO-max	+0.9159	+0.8103	+0.5398	+0.9036	
CPU-SLO-var	+0.8655	+0.8013	+0.5048	+0.7792	+0.9104

Table 4.6: Correlations between CPU-centric metrics.

	RAM-avg	RAM-max	RAM-var	RAM-SLO-avg	RAM-SLO-max
RAM-max	+0.8187				
RAM-var	+0.6287	+0.9290			
RAM-SLO-avg	+0.6368	+0.8095	+0.7599		
RAM-SLO-max	+0.6473	+0.8175	+0.7420	+0.9709	
RAM-SLO-var	+0.5504	+0.8340	+0.8355	+0.9527	+0.9466

Table 4.7: Correlations between RAM-centric metrics.

FET is inherently only affected marginally by the autoscaling setup, or variations in FET are generally too small to significantly impact the total time a request spends in the system, or the way the FaaS-Sim simulator functions internally causes this behavior. In either case, FET appears to not be a particularly well-suited metric for the given setting.

Table 4.6 shows the correlation values between the metrics pertaining to CPU utilization. The highest correlation among them can be found between CPU-SLO-avg and CPU-avg with a coefficient of +0.9484, indicating that the raw utilization and the SLO violation rate convey roughly the same information when averaged over all nodes. One can observe a high correlation between the respective maxima and the variance variations, with coefficients of +0.9363 and +0.9104 for the variations based on raw utilization and SLOs, respectively. This indicates that it is likely that high utilization maxima are primarily caused by uneven distribution of load on the nodes. There is a comparatively low correlation between CPU-var and its SLO-based counterpart at only +0.5048. This may indicate that an unequally distributed load does not necessarily mean that nodes with higher loads violate their target utilization. The mean and maximum values have correlation coefficients of +0.8885 and +0.9036 for the raw utilization- and SLO-based variations, respectively, indicating a rather large overlap in information.

RAM-centric metrics, listed in Table 4.7, show mostly similar relationships with a few exceptions. For one, the correlation coefficient between RAM-avg and RAM-SLO-avg is only +0.6368 and therefore much lower than its CPU counterpart. This is most likely attributed to the fact that the utilized test function is rather CPU-heavy and uses comparably few memory resources. Therefore, SLO violations of the memory constraints become rather rare, significantly lowering the overall SLO violation rate. Similar differences can also be observed for the other SLO-based variations as well, which is most likely attributable to the same reason.

Metric	avg	max	var	SLO-avg	SLO-max	SLO-var
Corr. Coeff.	+0.9987	+0.9578	+0.9891	+0.4357	+0.5222	+0.4221

Table 4.8: Correlations between respective RAM and CPU utilization metrics.

	CvE-rate	ZC-rate
RTT	+0.7864	+0.8429
RTT@90	+0.8533	+0.8684
RTT@95	+0.8619	+0.8607
RTT@99	+0.8373	+0.8511
RTT-SLO-rate	+0.8892	+0.9742
SD-rate	+0.7043	+0.8589

Table 4.9: Correlations between timing-based metrics and metrics pertaining to processing location.

The two replica-based metrics, rep and rep-rate, show a high correlation with a coefficient of +0.9891, indicating little difference in choosing one over the other and showing no indication that using both for assessing quality provides any benefit.

Correlations Between Groups

SD-rate strongly correlates with the overall RTT of requests. For example, SD-rate correlates with RTT with a coefficient of +0.8685 and with RTT-SLO-rate with a coefficient of +0.9039. This shows that the main contributors to how long a request spends in the system are not related to the actual processing of requests but rather to platform overhead such as queue times and latencies from forwarding requests to different zones.

Table 4.8 lists the correlation coefficients between respective RAM and CPU utilization metrics. It is shown that the variations using raw utilization heavily correlate, while the SLO-based variations show a low correlation. Therefore, it is reasonable to assume that the metrics that observe raw utilization are not affected by the type of workload. However, the SLO-based variations seem to develop differently depending on whether the executed functions are more CPU- or memory-intensive, which is to be expected.

Table 4.9 shows the relationship between metrics that capture where requests are processed and timing metrics such as RTT and SD. As expected, all listed pairs are highly correlated, indicating that processing requests in different zones from the origin or offloading entirely to the cloud has a large impact on response times. The biggest measured impact was between RTT-SLO-rate and ZC-rate with a correlation coefficient of +0.9742, indicating that processing requests in a different zone is one of the main contributing factors to requests missing their deadlines.

Table 4.10 shifts the analysis of processing-location-related metrics to a comparison with resource-focused metrics. Since the synthetic test function used was CPU-intensive, the

	CvE-rate	ZC-rate
CPU-avg	−0.1055	−0.0469
CPU-max	−0.1359	−0.0511
CPU-var	−0.0592	+0.0515
CPU-SLO-avg	−0.1063	−0.0594
CPU-SLO-max	−0.3153	−0.2368
CPU-SLO-var	−0.2780	−0.2482

Table 4.10: Correlations between CPU-based metrics and metrics pertaining to processing location.

analysis focuses on CPU utilization metrics only. In general, these metrics all show a weak to moderate negative correlation except the very weakly positively correlated pair of CPU-var and ZC-rate. Hence, it is reasonable to assume that the more requests are processed on the edge, the more strain is put on the limited resources in that zone, leading to higher utilization compared to when the load is more fairly distributed to other edge zones or the cloud. CPU-var seems to be the least impacted by the processing location. The SLO-based counterpart CPU-SLO-var is more impacted. The processing location appears to have the highest impact on CPU-SLO-max. Cloud processing seems to particularly affect this metric with correlation coefficients of −0.3153, the highest observed in the table. The raw utilization metrics are also impacted but to a lesser extent. CvE-rate and ZC-rate show generally the same pattern, but the impact of zone crossing seems to be slightly less pronounced than that of cloud offloading. All of this supports the assumption that offloading is a useful strategy to prevent the over-utilization of edge resources.

Analysis of Metric Pairs of Interest

In general, regarding correlation values and their significance for finding a representative set of metrics to describe system quality, the following assumptions are made:

1. Pairs of metrics that have a correlation coefficient far below 0 indicate competing aspects of the system and are worth including in a quality function.
2. Pairs of metrics that have a correlation coefficient close to 0 indicate mostly independent aspects of the system and are worth including in the quality function.
3. Pairs of metrics that have a high correlation coefficient indicate similar aspects of the system and should be avoided.

According to these assumptions, the top 20 pairs of metrics that fall into each of these categories are gathered and compiled in Table 4.11. Notably, since high correlation is to be expected between metrics of the same group, these pairs were filtered out of the column "Most Correlated", as there would be little value in taking a closer look at them. The list of

Rank	Most Anticorrelated			Least Correlated			Most Correlated		
	Metric	Metric	Corr.	Metric	Metric	Corr.	Metric	Metric	Corr.
1	CvE-rate	CPU-SLO-max	−0.3153	RTT@95	RAM-max	+0.0004	CPU-avg	RAM-avg	+0.9987
2	CvE-rate	CPU-SLO-var	−0.2780	cost	rep-rate	−0.0012	RTT-SLO-rate	ZC-rate	+0.9742
3	FET@99	CPU-SLO-max	−0.2595	FET-SLO-rate	CPU-SLO-max	+0.0020	CPU-var	RAM-var	+0.9642
4	ZC-rate	CPU-SLO-var	−0.2482	FET-SLO-rate	CPU-SLO-var	−0.0032	CPU-var	RAM-max	+0.9630
5	ZC-rate	CPU-SLO-max	−0.2368	RTT@90	CPU-max	+0.0039	CPU-max	RAM-max	+0.9578
6	RTT@95	CPU-SLO-max	−0.2347	SD-rate	CPU-SLO-avg	+0.0045	CPU-SLO-avg	RAM-avg	+0.9393
7	RTT@99	CPU-SLO-max	−0.2260	RTT@99	CPU-max	+0.0055	CPU-avg	rep-rate	+0.9226
8	FET@90	CPU-SLO-max	−0.2256	cost	CPU-var	+0.0074	CPU-SLO-max	rep-rate	+0.9221
9	RTT-SLO-rate	CPU-SLO-var	−0.2251	FET-SLO-rate	RAM-max	−0.0074	CPU-avg	rep	+0.9196
10	FET@99	CPU-SLO-var	−0.2238	cost	RAM-var	+0.0077	RAM-avg	rep-rate	+0.9191
11	FET@95	CPU-SLO-max	−0.2212	RTT	CPU-avg	+0.0096	CPU-SLO-max	RAM-avg	+0.9172
12	RTT@95	CPU-SLO-var	−0.2190	cost	CPU-max	+0.0103	CPU-SLO-max	rep	+0.9164
13	FET@99	rep-rate	−0.2167	FET-SLO-rate	CPU-max	−0.0103	RAM-avg	rep	+0.9154
14	CvE-rate	CPU-SLO-var	−0.2166	RTT@95	RAM-SLO-max	+0.0139	CPU-SLO-var	rep	+0.9088
15	RTT@90	CPU-SLO-max	−0.2164	RTT-SLO-rate	CPU-max	−0.0146	CPU-SLO-var	rep-rate	+0.9080
16	CvE-rate	rep-rate	−0.2121	RTT-SLO-rate	RAM-max	+0.0172	RTT-SLO-rate	SD-rate	+0.9040
17	RTT@90	CPU-SLO-var	−0.2108	RTT	RAM-avg	+0.0174	CPU-max	RAM-avg	+0.8991
18	CvE-rate	rep	−0.2094	RTT@95	CPU-max	−0.0176	CPU-SLO-avg	rep-rate	+0.8953
19	RTT-SLO-rate	CPU-SLO-max	−0.1983	cost	RAM-max	−0.0183	CPU-SLO-avg	rep	+0.8918
20	FET@99	rep	−0.1914	FET-SLO-rate	RAM-SLO-avg	−0.0196	RTT-SLO-rate	CvE-rate	+0.8892

Table 4.11: Top 20 list of the different interest groups of metric pairs and their correlation coefficients.

low correlation pairs is dominated by CPU-SLO-max and CPU-SLO-var, with the highest magnitudes observed regarding edge vs. cloud processed requests followed by various timing-based metrics. The column of independent pairs is not clearly dominated by any group of metrics. However, there are many entries containing a timing-based metric and a resource-based metric. This may indicate that there is not too much impact on timings when performing well-tuned request offloading. The filtered column of most correlated pairs is also rather varied but features a notable amount of pairs including a CPU and a RAM metric. This indicates that despite there being an imbalance between the CPU and the RAM requirements of the test function, there still exists a relationship between the according resource utilization metrics. Furthermore, replica-based metrics appear rather often, indicating that they, in general, do not express much unique information not found among other metrics.

4.3.4 Key Performance Metrics and Quality Function Formulation

To arrive at a reasonable set of metrics based on the data gathered, the given metrics and correlations between them were transformed into a graph structure, constructed as follows. The graph has one vertex per metric, labeled with the corresponding metric's name. The vertices are fully connected with undirected weighted edges, where each weight represents the correlation coefficient between the connected vertices. Subsequently, the FET-based metrics were removed from the graph. As stated above, the data collected throughout the experiments showed that FET is only marginally impacted by the choice of autoscaler parameters. Including metrics based on FET as an optimization goal therefore adds little value. The remaining graph was then used to extract a subgraph of seven nodes, which includes all edges between them, such that the total weight of the edges in the subgraph is minimized. This should ideally lead to a vertex set that matches the criteria

Metrics	Combined Weight
RTT-SLO-rate, CvE-rate, ZC-rate, cost, CPU-SLO-max, CPU-SLO-var, RAM-SLO-var	3.5561
CvE-rate, ZC-rate, SD-rate, cost, CPU-SLO-max, CPU-SLO-var, RAM-SLO-var	3.5792
RTT-SLO-rate, CvE-rate, SD-rate, cost, CPU-SLO-max, CPU-SLO-var, RAM-SLO-avg	3.6393
RTT-SLO-rate, CvE-rate, ZC-rate, cost, CPU-SLO-max, CPU-SLO-var, RAM-SLO-avg	3.6552
CvE-rate, ZC-rate, SD-rate, cost, CPU-SLO-max, CPU-SLO-var, RAM-SLO-var	3.6576

Table 4.12: Top five metric-correlation subgraphs of seven metrics with the lowest combined edge weights.

established for a favorable set of target metrics. The graph's size of seven nodes was chosen to allow for a heterogeneous mix of metrics while maintaining a certain level of simplicity for further analysis.

Table 4.12 lists the vertices of the top five subgraphs obtained using the described method. It becomes immediately apparent that SLO-based resource metrics dominate the list, which also mirrors their prevalence in the relevant columns of Table 4.11. Additionally, two metrics that are present in all subgraphs are CvE-rate and cost, which can be rationalized by them being roughly opposites of one another. CvE-rate improves when more requests are processed on the edge, which is considered more expensive in the AWS Lambda cost model. Furthermore, cost and resource metrics seem to be rather independent of each other, as can be seen in the central row of Table 4.11. The last two metrics are always a choice among either ZC-rate, RTT-SLO-rate, and SD-rate.

Based on these results, the following metrics were chosen for future optimization efforts:

- RTT-SLO-rate,
- cost,
- CvE-rate,
- ZC-rate,
- CPU-SLO-max,
- CPU-SLO-var,
- RAM-SLO-max, and
- RAM-SLO-var.

The set of metrics is based on the subgraph with the lowest weight and is extended with RAM-SLO-max to achieve parity between CPU- and RAM-based metrics, allowing the quality formulation to be used for both compute- and memory-heavy workloads.

When looking at multi-objective optimization, these eight metrics can be used directly without any further transformation. Furthermore, they shall serve as KPIs of edge-cloud orchestration for evaluation going forward.

For single-objective optimization, these metrics need to be weighed, normalized, and combined. This leads to the final formulation of orchestration quality for the purpose of a quality function used for optimization being:

$$\begin{aligned} quality = & \frac{1}{5} * \textit{RTT-SLO-rate} + \frac{1}{5} * \textit{norm-cost} + \\ & \frac{2}{15} * \textit{CvE-rate} + \frac{1}{15} * \textit{ZC-rate} + \\ & \frac{1}{10} * \textit{CPU-SLO-max} + \frac{1}{10} * \sqrt{\textit{CPU-SLO-var}} + \\ & \frac{1}{10} * \textit{RAM-SLO-max} + \frac{1}{10} * \sqrt{\textit{RAM-SLO-var}} \end{aligned} \tag{4.2}$$

The variance metrics were transformed to their respective standard deviation because the observed variance values were rather low in magnitude and would otherwise not have a significant impact on the overall score. Additionally, the units now match across the resource-based metrics. Furthermore, the cost metric had to be normalized into a new metric: norm-cost. This is achieved by dividing the actual cost of a run by the maximum possible cost. The maximum can be calculated by assuming that all requests were processed on the edge instead of the cloud, which is more expensive in the given cost model. Notably, this definition may not hold for other cost models that are not based on AWS Lambda@Edge.

With the normalizations and adaptions mentioned above, all metrics are now defined in a range between 0 and 1, where smaller values are more favorable. The total quality score is a weighted sum over the metrics, with weights, that are chosen in a way such that the same value range is preserved. Resource-based metrics are weighed less intensely, so both the CPU and RAM metrics contribute 1/5 to the total score each. CvE-rate and ZC-rate were also weighed in such a way that they together contribute 1/5 to the total score. However, CvE-rate was weighed more, because cloud offloading is generally less favorable than offloading to another edge zone.

Finally, it is worth noting that the given definition of quality is by far not the only valid one, and no claims are being made that it is in any way optimal. As described in Section 4.1, there are many challenges in defining a numeric value representing orchestration quality. The given definition is simply the result of the described methodology, which should result in a reasonably formulated quality function for further experimentation with optimization concerning autoscaler parameters.

CHAPTER 5

Autoscaler Configuration Optimization

This chapter covers the research activities performed to solve the introduced autoscaler parameter optimization problem. It begins with a formal problem definition for single and multi-objective scenarios with respect to a concrete autoscaling solution. Then, it outlines the challenges unique to this setting. Finally, it presents the implemented approach and, for each explored algorithm, describes adaptations made to fit the scenario, and highlights remaining open hyperparameters.

5.1 Problem Formalization

As previously mentioned, the optimization of orchestration parameters explored in this thesis is limited to the configuration of the autoscaling mechanism. Specifically, the given setting uses the pressure-based autoscaling approach introduced by Raith et al. [66]. Here, the static configuration amounts to two thresholds p_{max} and p_{min}. The upper threshold p_{max} controls how quickly the system deploys new function replicas when the demand on the system starts to increase. The higher this value, the higher the load needs to be before the autoscaler schedules new function replicas to be deployed. Inversely, the lower threshold p_{min} controls the downscaling of deployed function replicas. The higher this threshold, the quicker the system will remove running replicas again when demand shrinks. Such a pair of pressure values exists for each function and each deployed autoscaler, of which there can be multiple when looking at decentralized deployments.

Let $\mathcal{F}$ be the set of all deployable functions. Let $\mathcal{Z}$ be the the set of distinct zones. A zone is defined as a set of nodes for which a single local autoscaler is responsible. This includes edge and cloud zones. By assigning every unique combination of function and zone two pressure thresholds, we arrive at the set of all possible pressure assignments for

given functions and zones defined as

$$\begin{aligned}\mathcal{T} = \{((p_{min,1}, p_{max,1}, f_1, z_1), (p_{min,2}, p_{max,2}, f_2, z_1), \ldots, (p_{min,nm}, p_{max,nm}, f_n, z_m) | \\ n = |\mathcal{F}|, m = |\mathcal{Z}|, \\ \forall_{i \leq nm}[p_{min,i}, p_{max,i} \in \mathbb{R}], \\ \forall_{i \leq n}[f_i \in \mathcal{F}], \forall_{i \leq m}[z_i \in \mathcal{Z}], \\ \forall_{i \leq n, j \leq m}[\neg \exists_{i' \neq i, j' \neq j}[f_i = f_{i'} \wedge z_j = z_{j'}]]\}.\end{aligned} \tag{5.1}$$

The number of distinct pressure thresholds, and therefore the total dimensionality of the optimization problem amounts to

$$d = 2 * |\mathcal{F}| * |\mathcal{Z}|. \tag{5.2}$$

Even simple settings can cause the dimensionality of the problem to grow quickly. For example, when deploying three steps of a machine learning pipeline – data collection, training, and inference – across two edge and one cloud zone, the total number of thresholds, and therefore the dimensionality of the optimization problem, already reaches 18.

The quality of a set of parameters is measured by observing the total orchestration quality. Hence, obtaining data for deducing a quality measurement involves running a simulation. Low-level metrics are assumed to be extractable from the simulation's results. To this end, we formally define a function

$$sim : \mathcal{T} \times \mathcal{I} \times \mathcal{W} \mapsto \mathcal{M} \tag{5.3}$$

,where $\mathcal{I}$ is the set of possible descriptions of edge-cloud infrastructures that the simulator consumes, $\mathcal{W}$ is the set of possible workloads to feed the simulator with, and $\mathcal{M}$ refers to the set of 8-tuples representing each of the target metrics listed in Subsection 4.3.4. Conceptually, $sim(t, i, w)$ takes workload w, configures a given infrastructure i with the threshold values given by t and runs a simulation with these inputs. After the simulation, the relevant metrics are gathered and returned as the function's result.

For the sake of single-objective optimization, we further define the function

$$quality : \mathcal{M} \mapsto \mathbb{R}^+. \tag{5.4}$$

quality takes the metrics returned by *sim* and then normalizes and combines them as described in Equation 4.2.

Finally, there exist two constraints on the threshold values:

1. p_{max} and p_{min} both have to be in range [0, 1],
2. and for each pair, it needs to holds that $p_{max} > p_{min}$.

Based on these definitions, we can define the single-objective optimization problem of interest as follows: Given known *topology*, *workload*, functions $\mathcal{F}$ and zones $\mathcal{Z}$, we arrive at the following problem.

$$\begin{aligned} \min_{t \in \mathcal{T}} \quad & quality(sim(t, topology, workload)) \\ \text{s.t.} \quad & t_{i,p_{min}} \in [0,1] && i = 1 \dots |\mathcal{F} \times \mathcal{Z}| \\ & t_{i,p_{max}} \in [0,1] && i = 1 \dots |\mathcal{F} \times \mathcal{Z}| \\ & t_{i,p_{min}} < t_{i,p_{max}} && i = 1 \dots |\mathcal{F} \times \mathcal{Z}| \end{aligned} \tag{5.5}$$

For the multi-objective counterpart, we can simply omit the *quality* function, as *sim* already returns a tuple of the target metrics. Therefore, we arrive at the following.

$$\begin{aligned} \min_{t \in \mathcal{T}} \quad & sim(t, topology, workload)) \\ \text{s.t.} \quad & t_{i,p_{min}} \in [0,1] && i = 1 \dots |\mathcal{F} \times \mathcal{Z}| \\ & t_{i,p_{max}} \in [0,1] && i = 1 \dots |\mathcal{F} \times \mathcal{Z}| \\ & t_{i,p_{min}} < t_{i,p_{max}} && i = 1 \dots |\mathcal{F} \times \mathcal{Z}| \end{aligned} \tag{5.6}$$

5.2 Problem Characteristics

Based on the given definition, some remarkable characteristics of the problem at hand are now outlined and elaborated.

As is apparent from the use of the *sim* function, evaluating a single set of pressure thresholds involves running a simulation. Hence, the quality function to be optimized is not differentiable, eliminating the possibility of using mathematical optimization techniques, such as gradient descent. Furthermore, experiments conducted with FaaS-Sim [65], the specific simulator used in the implementation of the proposed scheme, have shown that there are slight fluctuations in the results obtained from simulation runs using the same input. In an experimental setting, running the simulation with the same inputs 10 times resulted in quality scores that produced a distribution with a standard deviation of 0.0065. This further cements the need for heuristic optimization techniques.

In general, running an entire infrastructure simulation as part of a quality function evaluation can be considered a costly operation that greatly limits optimization approaches to those that emphasize quick convergence and efficiency.

The constraints imposed on the pressure pairs by the given setting are rather simple. Limiting the range of parameters is usually supported by most established optimization algorithms. The remaining constraint cleanly cuts the search space in half for each pressure pair. While this does lead to a heavily constrained search space, the topology of the valid space is not very complex. This allows simple constraint handling mechanisms to be effectively used because no special attention needs to be paid to a complexly constrained environment. Hence, the *sim* function can be implemented in a way that skips running the simulation if a constraint is violated and instead returns a value immediately. Two different approaches are possible to arrive at a value for invalid parameters.

1. The $reject_{max}$ approach returns a fixed value, considerably above 1, which is the maximum possible quality for an actual run of the simulation.
2. The $reject_{rel}$ approach scores each set of invalid parameters based on how much the pairs violate the constraint, but always higher than a valid simulation run. The idea here is that this approach could, in theory, guide the optimization scheme towards valid parameters more efficiently.

Similarly, some of the investigated approaches may benefit from being initialized with a set of reasonable, but not necessarily optimal parameters. For example, one can assume that a parameter set where all minimum thresholds are set to 0.3 and all maximum thresholds are set to 0.7 represents at least a reasonable choice. However, some schemes might perform better when initialized fully randomly.

Another specialty of the outlined problem is that while most simulation runs finish in less time than that which is simulated, given highly suboptimal parameters, the simulation may take considerably longer to finish. Since the overall quality function of the optimization acts as a black box, there is no way to tell up-front whether a simulation run will finish fast or slow, making an evaluation even costlier. This creates a challenging setting, where the ability of an optimization scheme to quickly arrive at a set of decently good parameters becomes especially important. Furthermore, the algorithm's suitability for parallel processing becomes non-trivial. This can be considered a problem specific to the utilized simulator. However, the emphasis on parallelizability of optimization algorithms likely also applies when using different simulators.

5.3 Implemented Approach

To enable the evaluation of different metaheuristics in the given setting, an experimental setup was created where each of the six algorithms described in Section 2.4 was implemented using Python. Figure 5.1 gives an overview of the approach that was taken to implement the scheme originally proposed in Section 1.2. Each algorithm was abstracted so that they share a common interface. An optimization algorithm takes a simulation

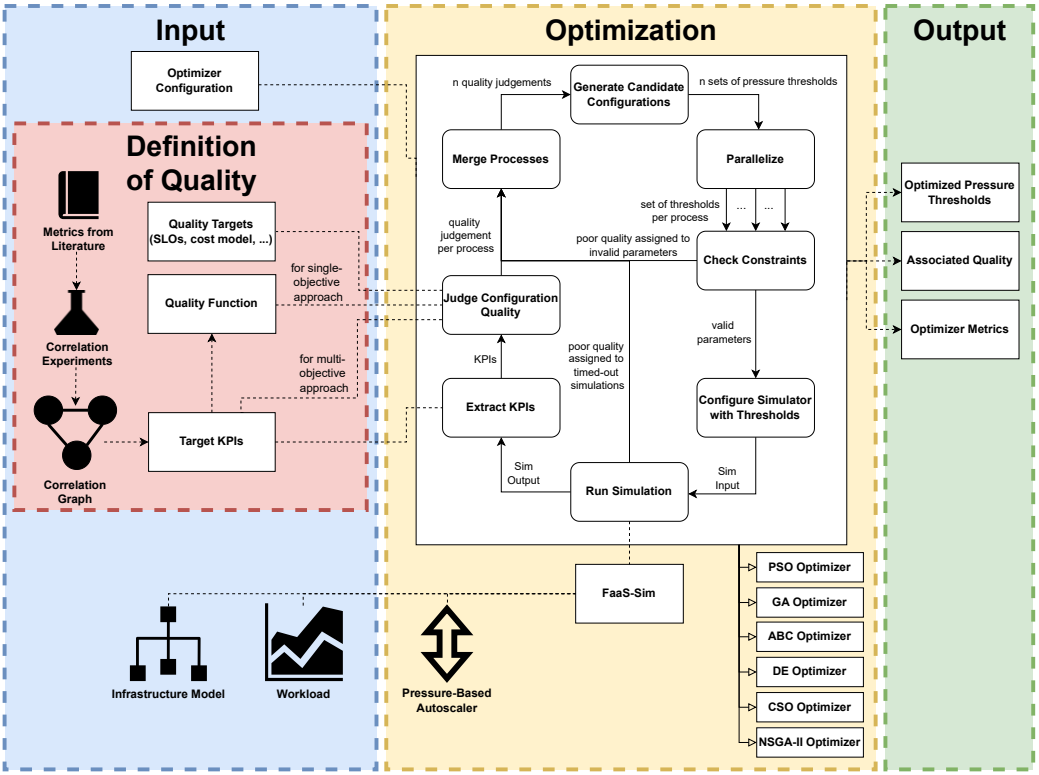

Figure 5.1: Schematic overview of the implemented approach.

setup consisting of an infrastructure and a workload model, the chosen definition of quality, and an optimizer-specific configuration, which includes the algorithm's hyperparameters, as input. It then runs the optimization and returns the best pressure thresholds, the associated quality score, and optimizer metrics useful for evaluation.

As already mentioned, the co-simulation-based FaaS-Sim edge-cloud simulator [65] is used in the implementation of the defined *sim* function. FaaS-Sim requires the following configuration.

- An orchestration configuration. This was configured to use the pressure-based autoscaling approach.
- An infrastructure model. This aspect was kept open as it changes between scenarios.
- And a workload, in the form of a list of inter-arrival times of requests for each edge zone. This aspect is also left open for the specific optimization scenario.

FaaS-Sim provides runtime metrics as Pandas dataframes at the simulation's end. Extraction functions were implemented for each metric used in the defined *quality* function

introduced in Chapter 4. Scenario-specific aspects, such as SLOs, are passed as parameters. The KPIs are then aggregated according to the outlined weighted sum approach. Alternatively, for multi-objective approaches, the extracted target KPIs serve directly as the notion of quality. This design allows for easy modification of the quality function by implementing different extraction functions or aggregation approaches, emphasizing flexibility as previously motivated.

To mitigate the impact of long-running simulator executions, the quality function was implemented in a way that allows optimizers to pass multiple sets of pressure thresholds to it at once, which are then all evaluated in parallel. Python's built-in process parallelism capabilities were used to achieve this. This necessitates adaptations to some metaheuristics, which are detailed in Section 5.4.

To reduce the impact of isolated, excessively long simulations, a timeout mechanism was implemented. If a simulation exceeds three times the simulated duration in real time, it is aborted, and its corresponding autoscaler configuration is assigned the worst possible quality score.

The aforementioned constraint handling strategies $reject_{max}$ and $reject_{rel}$ were implemented for each metaheuristic regarding the inequality constraint. The choice of which to use was added as a hyperparameter for each algorithm. Bound constraint handling was implemented in algorithm-specific ways. Details are given in Section 5.4.

Analogously, the choice of random or predefined initialization was implemented for each algorithm and represents a further open hyperparameter. The exact strategy differs for each algorithm, which is elaborated in Section 5.4.

Each algorithm terminates after a set number of iterations. Although convergence- and stagnation-based approaches exist, this one was chosen as it allows for easy comparison between optimizers.

5.4 Implemented Optimization Algorithms

This section details the adaptations made to the canonical metaheuristics presented in Section 2.4. It provides justifications for parameter settings, discusses constraint handling and initialization strategies, and lists remaining hyperparameters requiring tuning.

5.4.1 PSO

The implemented PSO-based optimizer uses the *PySwarms* [37] toolkit for PSO optimization in Python. The toolkit is open source and publicly available on GitHub[1]. Even though the original author no longer maintains the repository, the facilities for allowing batched evaluations of quality functions make it a fitting choice for the outlined setting. Inspecting the code reveals that the batched processing is achieved by first calculating all

[1] https://github.com/ljvmiranda921/pyswarms

Parameter	Value Range	Effect
swarmsize (n)	[0, ∞]	Controls the number of particles. More particles lead to greater exploration, but result in more quality function evaluations.
personal influence. (c_1)	[0, 2]	The higher, the more drawn a particle will be to its own *pbest*.
social influence (c_2)	[0, 2]	The higher, the more drawn a particle will be to the *gbest*.
inertia (w)	[0, 1]	The higher, the more a particle will maintain its current heading.
random initialization	$\{True, False\}$	If $True$, the initial positions are randomly generated. If $False$, the initial positions are based off a predefined solution and only headings are randomized.
constraint strategy	$\{reject_{max}, reject_{rel}\}$	If $reject_{max}$, invalid solutions are penalized equally. If $reject_{rel}$, solutions are penalized relative to how far they are from the valid search space.

Table 5.1: Hyperparameters of the implemented PSO algorithm.

new positions of particles in each iteration and then running all evaluations at once. This delays the update of *gbest*, which may cause slightly slower convergence of the algorithm.

Particle positions can be initialized either fully randomly or to a set position. If the latter is the case, the velocities and headings are still kept random to allow for the necessary diversity among the population.

The toolkit offers algorithm-specific constraint handling options in the form of setting locations of particles that violate a constraint to either a random location or the last known valid location with an inverted heading. The latter was chosen for bound handling.

A complete list of open hyperparameters is given in Table 5.1. It comprises the swarmzize, various factors that are used for calculating a particle's new positions at each step, and the two problem-specific choices for initialization and constraint handling strategy.

5.4.2 GA

A version of an elitist GA was implemented. This choice was made because of the preference for faster convergence over exhaustive exploration due to a costly quality function. A steady-state GA was deemed fundamentally suboptimal, as it only allows for very poor parallelization. The implementation makes use of the *PyGAD* [18] toolkit, which is an open source Python library, publicly available on GitHub[2]. It supports a wide variety of established variations of genetic operators, offers facilities for parallel quality function evaluation, and is highly extensible. One of the most important factors for GA performance is the choice of the genetic encoding of the solution space. Unfortunately, it is not possible to encode the inputs of the quality function in this setting as a representative bit vector, which would be the favorable type of encoding for good GA performance [70]. Instead, a floating-point encoding is used, where each pressure threshold represents one chromosome. However, one can argue that this does satisfy the schema theory to a certain extent, as single pressure thresholds individually contribute to the overall performance and using them as a unit of operations may likely yield favorable results. Crossover was

[2]https://github.com/ahmedfgad/GeneticAlgorithmPython

Parameter	Value Range	Effect
population size (n)	[2, ∞]	Controls the number of individuals in a generation. More individuals can lead to better exploration, but come at the cost of more goal function evaluations.
crossover probability (cp)	[0, 1]	Controls the likelihood of each chromosome to be subjected to crossover.
mutation probability (mp)	[0, 1]	Controls the likelihood of each chromosome to be subjected to mutation.
mating parents (p)	[2, n]	Controls how many parents contribute genetic material to new offspring.
elitism count (ec)	[0, n]	Controls how many high-fitness individuals to keep in the population each generation.
selection strategy (ss)	$\{rws, sus, rank, tnmt\}$	Determines which parent selection strategy to use.
random initialization	$\{True, False\}$	If $True$, the initial population is randomly generated. If $False$, the initial generation is based on a predefined solution with random mutations applied.
constraint strategy	$\{reject_{max}, reject_{rel}\}$	If $reject_{max}$, invalid solutions are penalized equally. If $reject_{rel}$, solutions are penalized relative to how far they are from the valid search space.

Table 5.2: Hyperparameters of the implemented GA algorithm.

chosen to be carried out uniformly, meaning pressure thresholds are swapped around individually and randomly between mating parents, but do not switch their position regarding functions and zones. Mutation is performed by setting the pressure values to random values in a valid range. Furthermore, to transform the given optimization problem into a maximization problem, the fitness values are calculated as $1/quality$.

Individuals can be initialized in two ways, either fully randomly, or by taking a base set of thresholds and creating individuals based on it where the thresholds have been modified by adding or subtracting a random, but valid value from them.

The canonical GA does not come with a specific mechanism for handling constraints of its own. However, by choosing appropriate mutation and crossover operators, one might be able to avoid the generation of invalid individuals depending on the type of constraints. In the given implementation, the genetic operators were chosen in a way in which solution bounds will be respected. However, the remaining inequality constraint on pressure values still needs to be handled by the established penalty strategies.

A list of all hyperparameters of the implemented GA is given in Table 5.2. It comprises the population size, probabilities for mutation and crossover, the number of mating parents, the number of elitist individuals, the parent selection strategy, and the problem-specific choices of initialization and constraint handling strategies. Regarding the selection strategy, all options offered by PyGAD are considered, except the strategy that would turn the algorithm into a steady-state GA. It is apparent that a major drawback of using a GA for optimization is the large amount of hyperparameters for which a logical choice based on prior knowledge does not exist or is only hard to infer correctly. This creates the risk that even when tuning them, a sufficiently good set of hyperparameters may not be found in a reasonable time.

Parameter	Value Range	Effect
number of workers (w)	$[1, \infty]$	Controls the number of worker bees. More workers can lead to more diversity among explored solutions, but comes at the cost of more quality function evaluations.
number of onlookers (o)	$[1, \infty]$	Controls the number of onlooker bees. More onlookers can lead to better exploitation, but comes at the cost of more quality function evaluations.
limit (l)	$[1, \infty]$	Controls how quickly food sources are considered exhausted.
random initialization.	{True, False}	If $True$, the initial population is fully randomly generated. If $False$, the initial food sources will include one predefined solution.
constraint strategy	$\{reject_{max}, reject_{rel}\}$	If $reject_{max}$, invalid solutions are penalized equally. If $reject_{rel}$, solutions are penalized relative to how far they are from the valid search space.

Table 5.3: Hyperparameters of the implemented ABC algorithm.

5.4.3 ABC

To allow batched evaluation of quality functions, necessary for parallelization, the canonical ABC, presented in Subsection 2.4.3, was slightly adapted. In each iteration, the onlookers are assigned to workers first, based on the worker's current food source. Afterward, the limit counters of each worker are checked and scouting occurs if required. This allows for all new food sources generated during the iteration to be evaluated at once. The remaining steps of the algorithms remain the same. These changes should not have any major impact on the algorithm's behavior or convergence. The number of scout bees is kept at 1, as originally proposed by the authors. The optimization problem is transformed into a maximization problem, analogously to the GA.

The initial food sources have to be randomly initialized. Otherwise, the algorithm would immediately converge, as diversity among food sources is needed for the generation of candidate solutions. To guide the optimization towards the area of decent configurations, a single predefined food source, already known to perform decently, can be injected into the initial population.

The algorithm does not come with a built-in constraint handling mechanism. For bound constraints, violating values are simply set to the closest bound. This was done because only one component in a food source's position is altered each time a new candidate is generated. Rejecting the value change would therefore lead to the same vector as the original food source and would be of no benefit.

Open hyperparameters are listed in Table 5.3. The numbers of workers and onlookers are kept open and may differ. In addition to those and the problem-specific choices for initialization and constraint handling strategy, there is only the limit l that remains open, leading to a comparably small set of hyperparameters.

Parameter	Value Range	Effect
population size (n)	$[4, \infty]$	Controls the number of individuals. More individuals can lead to higher diversity, which promotes exploration, but comes at the cost of more quality function evaluations.
crossover probability (cp)	[0, 1]	Controls the likelihood of vector positions being adapted during crossover.
differential weight (dw)	[0, 2]	Controls magnitude of changes to vector positions.
random initialization	$\{True, False\}$	If $True$, the initial population is fully randomly generated. If $False$, the initial population will include one predefined solution.
constraint strategy	$\{reject_{max}, reject_{rel}\}$	If $reject_{max}$, invalid solutions are penalized equally. If $reject_{rel}$, solutions are penalized relative to how far they are from the valid search space.

Table 5.4: Hyperparameters of the implemented DE algorithm.

5.4.4 DE

The implemented version of DE is mostly identical to the one presented in Subsection 2.4.4 with the only difference that in each iteration, all new candidate vectors are generated first and then evaluated in a batched manner. This does have the effect that the replacement of individuals also happens batched as opposed to one by one, possibly slowing down algorithmic convergence. However, the performance gains achieved from parallel quality function executions are considered more impactful.

Analogously to ABC, individuals need to be randomly generated to prevent immediate convergence. Hence, the nonrandom initialization strategy also injects one predefined threshold configuration among the initial vectors.

The bound constraints are handled by simply rejecting mutated vector components that violate the bounds. Since with DE multiple parts of a vector are likely altered in a step, this is considered a better alternative to setting violating components to the bound values.

The remaining open control parameters are listed in Table 5.4. In addition to the choices for strategies shared among all implemented algorithms, it contains the population size and the two parameters driving the crossover mechanism. The list is considerably shorter than that of the GA, even though both are based on the same evolution-inspired principles.

5.4.5 CSO

The implemented version of CSO differs from the one presented in Subsection 2.4.5. The order of operations has been slightly shifted. However, this should not have a major impact on the overall flow of the algorithm. In the implemented version, nests can have either an evaluated quality or an unknown quality. For one, this allows for batched quality function evaluation. On the other hand, since the nest locations are revisited in later iterations, this prevents unnecessary reevaluations of the quality function. After

Parameter	Value Range	Effect
number of nests (n)	$[1, \infty]$	Controls the number of solutions considered in each iteration. Authors of original publication recommend 20.
abandonment rate (p_a)	[0, 1]	Controls the fraction of nests abandoned each iteration. A higher value promotes more exploration at the cost of exploitation.
initialize randomly	$\{True, False\}$	If $True$, the initial population is fully randomly generated. If $False$, the initial nests will include one predefined solution.
constraint strategy	$\{reject_{max}, reject_{rel}\}$	If $reject_{max}$, invalid solutions are penalized equally. If $reject_{rel}$, solutions are penalized relative to how far they are from the valid search space.

Table 5.5: Hyper parameters of the implemented CSO algorithm.

performing the cuckoo's Lévy flight, all unknown costs are evaluated, including the one associated with the cuckoo's new location, in a batched manner. Afterward, replacement happens as usual, with all newly generated nests assigned an unknown quality. The original paper does not specify that the generation of nests during replacement needs to happen via Lévy flights. However, it was decided to implement it this way based on the adaptations introduced by Walton et al. [89]. The parameter α controlling the step size has been set to 0.01 to complement the threshold value range of [0, 1].

It makes little sense not to initialize the nests randomly. Therefore, the same approach regarding initialization, already introduced for ABC and DE, was chosen, where a known configuration may be inserted among the initial nests.

The bound handling approach proposed by Walton et al. [89] was implemented, where constraint-violating parameters are simply not modified at all, meaning that some parameters of the original solution may carry over into the new one.

Open hyperparameters of CSO are listed in Table 5.5. It is often mentioned as a strength of CSO that it only has a single control parameter p_a. However, that is only true if one does not consider the number of nests n as a control parameter. Yang and Suash Deb [97] recommend using 20 nests for any problem and present evidence that neither the number of nests nor the choice of p_a, severely impact the algorithm's convergence behavior. The additional problem-specific hyperparameters extend this list even further.

5.4.6 NSGA-II

A version of NSGA-II was implemented using the *PyGAD* library, which was also used for the regular single-objective GA.

The hyperparameters of NSGA-II are also, for the most part, the same as for the GA. A list of them is given in Table 5.6. The main difference is that the options for the selection strategy have been reduced to a choice between a pure sorting-based selection and tournament selection using Pareto rank and crowding distance. Additionally, because the algorithm is elitist by design, the elitism count does not need to be explicitly defined.

Parameter	Value Range	Effect
population size (n)	[2, ∞]	Controls the number of individuals in a generation. More individuals can lead to higher diversity, which promotes exploration, but comes at the cost of more quality function evaluations.
crossover probability (cp)	[0, 1]	Controls the likelihood of each chromosome to be subjected to crossover.
mutation probability (mp)	[0, 1]	Controls the likelihood of each chromosome to be subjected to mutation.
mating parents (p)	[2, n]	Controls how many parents contribute genetic material to new offspring.
selection strategy (ss)	$\{sorting, tnmt\}$	Determines which selection strategy to use.
random initialization	$\{True, False\}$	If $True$, the initial population is randomly generated. If $False$, the initial generation is based on a predefined solution with random mutations applied.
constraint strategy	$\{reject_{max}, reject_{rel}\}$	If $reject_{max}$, invalid solutions are penalized equally. If $reject_{rel}$, solutions are penalized relative to how far they are from the valid search space.

Table 5.6: Hyperparameters of the implemented NSGA-II algorithm.

CHAPTER 6

Evaluation of Selected Optimization Approaches

This chapter evaluates the six metaheuristics that were implemented in terms of their suitability for optimizing static configuration parameters of an edge-cloud autoscaling solution. First, the setting used for evaluation purposes is described. Subsequently, values for all open hyperparameters of the algorithms are systematically chosen and motivated. Finally, the results of the conducted experiments are presented with the goal of being used as a basis for making recommendations concerning the evaluated algorithms.

6.1 Experimental Setup

This section outlines the setup of the experiments that were used to evaluate optimizer performance, including the utilized infrastructure, function, performance goals, and workload.

6.1.1 Infrastructure

The experiments use a small-scale smart city edge-cloud infrastructure depicted in Figure 6.1, comprising a central cloud zone and eight edge zones. Each zone employs decentralized autoscaling and scheduling via local controllers. There are two types of nodes in the system. *XEON* nodes are modeled after a system running an Intel Xeon E-2224 processor and represent strong nodes. They are equipped with 4 CPU cores based on the x86 architecture at a clock speed of 3.4 GHz and 16 GB of RAM. *NX* nodes are modeled after an embedded NVIDIA Jetson Xavier NX system and represent weak nodes. They have 6 CPU cores at a clock speed of 1.9 GHz based on the ARM architecture and 8 GB of RAM. Persistent storage is not modeled in the given scenario, neither are GPU capabilities of the nodes. There are three types of zones. *IoT-Box* zones model small

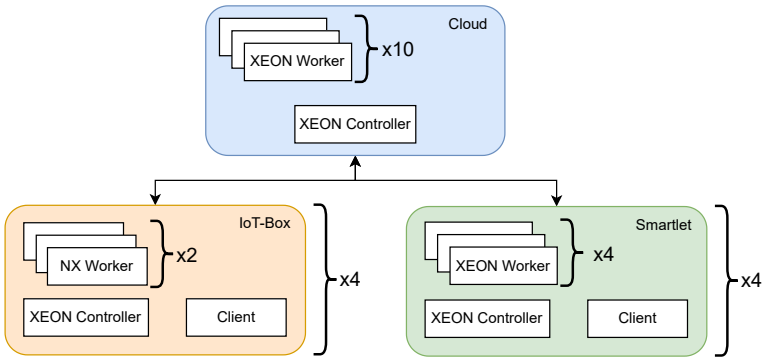

Figure 6.1: Infrastructure used for optimizer experiments.

clusters of low-power nodes consisting of two *NX* nodes and one *XEON* node, which also functions as the controller for the zone. *Smartlet* zones model small-scale cloudlets that consist of five *XEON* nodes, including one, which also functions as the zone's controller. The *Cloud* zone represents a cloud data center. It consists of 11 *XEON* nodes, which also include one that functions as the zone's local controller and the global master node for the whole deployment. Edge zones 1 to 4 are *IoT-Box* zones. Edge zones 5 to 8 are *Smartlet* zones. Each edge zone includes a *Client* node that simulates a gateway for user requests.

6.1.2 Function

The clients of the system invoke a single deployable function. This function simulates an AI speech inference task. Execution times on the different nodes are simulated based on measurements made on the real devices that the simulated nodes model. The function's image allocates a fixed amount of 262144 KB of memory and 2 CPU cores on deployment. It is designed to be a realistic, CPU-intensive task, and requires disproportionately more compute resources than memory. The autoscaling interval is 5 seconds.

6.1.3 Performance Goals and Parameters

For defining the KPIs and the quality score introduced in Subsection 4.3.4 the function is assumed to have an RTT-based SLO of 3 seconds. Furthermore, resource-oriented metrics are calculated with an SLO of 80 % utilization for both CPU- and RAM-based metrics. The cost model uses AWS Lambda@Edge [73] pricing as of 03.02.2025: 0.6 $ per million requests processed, 0.0000500100 $ per GB-second at the edge, and 0.0000166667 $ per GB-second in the cloud.

6.1.4 Workload

Requests are generated differently for each edge zone. A visualization of the respective RPS profiles is given in Figures 6.2 and 6.3. The light blue lines show the exact RPS

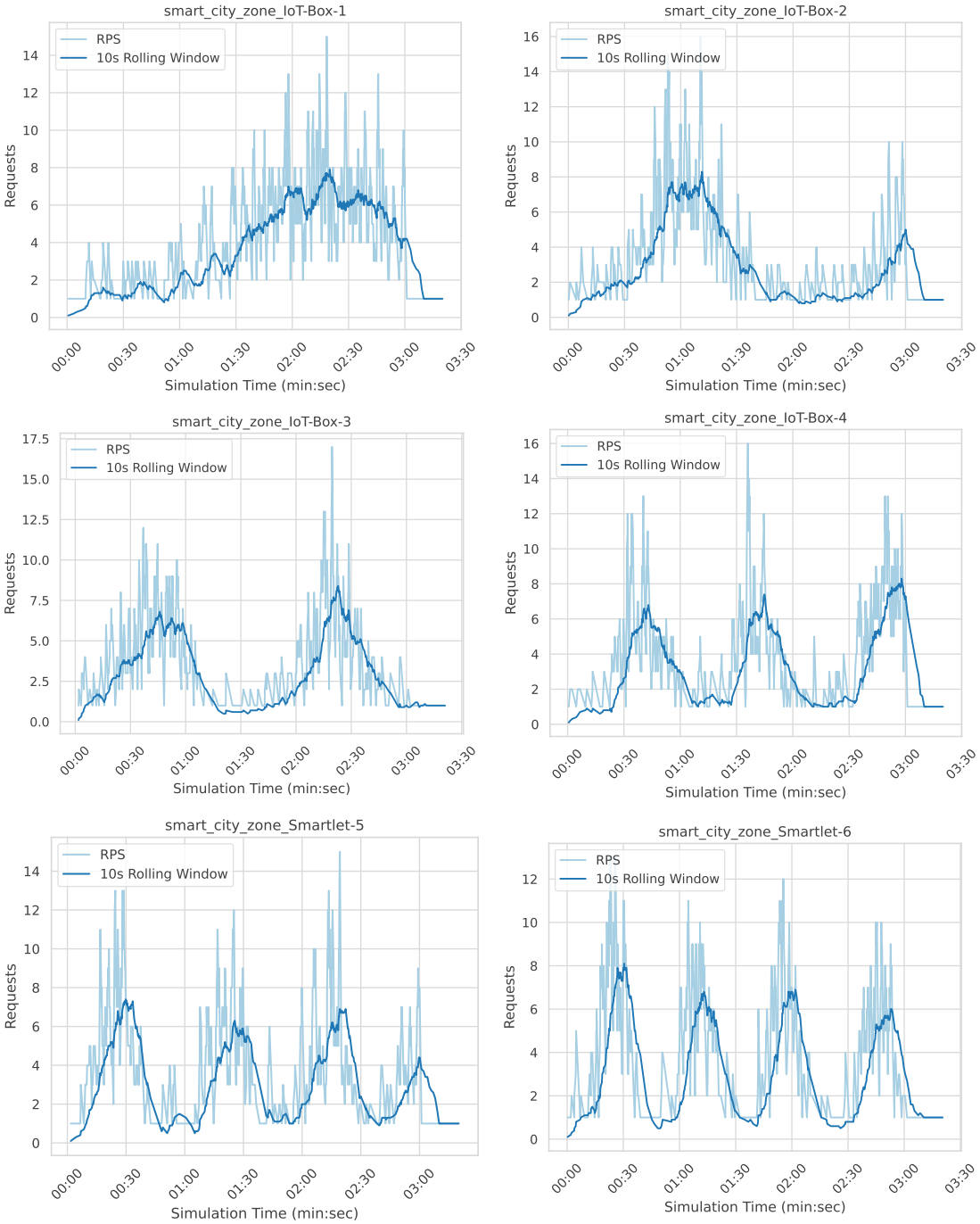

Figure 6.2: Request profiles used in optimizer experiments.

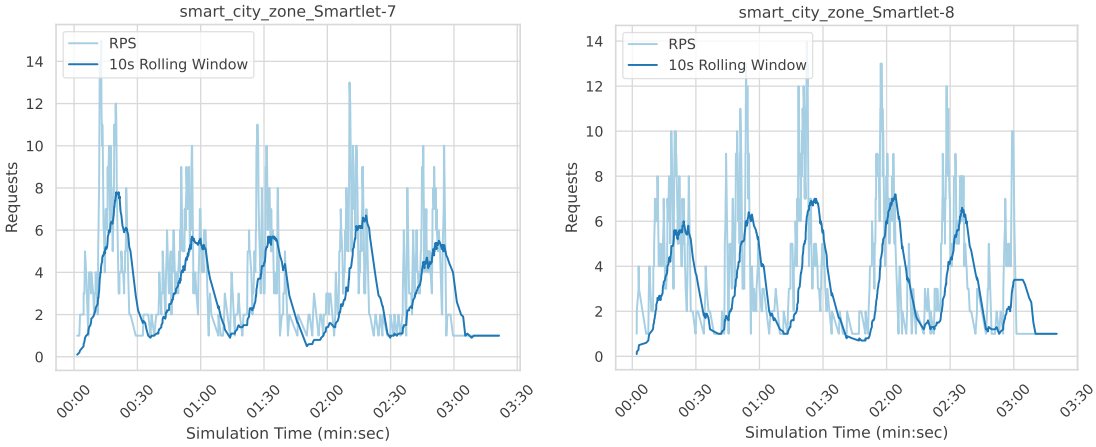

Figure 6.3: Request profiles used in optimizer experiments continued.

values, while the dark blue line gives a broader perspective on the generated requests by showing a 10-second rolling average. The simulated scenario is exactly 3 minutes long followed by a tail padding of 20 seconds where no new requests are generated. This is done to allow even a suboptimally configured platform to fully process most requests in the simulated time window. Each request profile is modeled to mimic a sine wave displaying random fluctuations, with each zone being assigned a different frequency that increases with the zone id. The motivation behind this is based on the fact that a recent analysis of 85 billion user requests processed by 5 different data centers hosting a serverless platform operated by Huawei [38] shows that RPS curves often display periodic patterns. Spikes in the middle of generally low usage are rather rare, and so are constant loads. The increase in frequency was implemented to introduce some variation into the profiles without diverging from the overall sine wave shape. The number of requests was chosen so that combining the average number of requests across all zones roughly equals 80 % of the infrastructure's capacity if there were no platform overhead.

6.2 Hyper Parameter Tuning

This Section is concerned with arriving at suitable configurations of the hyperparameters for each evaluated metaheuristic regarding the outlined scenario. First, the approach that was followed is described. Subsequently, choices for quality indicators for optimizer performance in this setting are presented and motivated. The rationale that was followed for selecting hyperparameter values is presented. The section concludes with a list and rationalization of the choices made.

6.2.1 Approach

It would be highly infeasible to run any form of automatic HPO for the scenario outlined in Section 6.1. The setting is too large in terms of the time it takes to fully simulate one run.

Algorithm	Parameter	Value Range
PSO	n	[5, 30]
	c1	[0.1, 2.0]
	c2	[0.1, 2.0]
	w	[0.4, 0.9]
	init rand	{True, False}
	cons strat	$\{reject_{max}, reject_{rel}\}$
GA	n	[10, 30]
	cp	[0.1, 0.9]
	mp	[0.1, 0.9]
	p	[2, 10]
	ec	[1, 5]
	ss	{rws, sus, rank, tnmt}
	init rand	{True, False}
	cons strat	$\{reject_{max}, reject_{rel}\}$
ABC	w	[1, 20]
	o	[1, 20]
	l	[1, 5]
	init rand	{True, False}
	cons strat	$\{reject_{max}, reject_{rel}\}$
DE	n	[4, 30]
	cp	[0.5, 1.0]
	dw	[0.0, 2.0]
	init rand	{True, False}
	cons strat	$\{reject_{max}, reject_{rel}\}$
CSO	n	
	p_a	[0.1, 0.9]
	init rand	{True, False}
	cons strat	$\{reject_{max}, reject_{rel}\}$

Table 6.1: Value ranges used to generate hyperparameter configurations.

Therefore, the experiments use the smaller-scale infrastructure and the deployable function of Subsection 4.3.2 introduced for the metric correlation experiments. To consider multiple different possible scenarios and increase statistical relevance, for each configuration of hyperparameters, five different workloads are optimized for 30 iterations each. The chosen workloads are among those also introduced for the correlation experiments in Chapter 4. They comprise *Sine Constant Mixed*, *Sine*, *Spikes Constant Mixed*, *Spikes*, and *Zone Switch*.

Traditionally, hyperparameter tuning can be done via a grid search, by using another heuristic algorithm, or by performing a random search [8]. Due to the characteristic of long-running quality function executions, only the last option is viable for the given scenario, even in the reduced setting. To this end, 20 hyperparameter configurations were generated for each algorithm, based on the ranges given in Table 6.1. While the data's representativeness regarding hyperparameter impact could be questioned, this limitation is unavoidable given the constraints of the setting. Further analysis proceeds on a best-effort basis.

6.2.2 Optimizer Performance Indicators

When evaluating how well an optimization algorithm performs under a given set of hyperparameters, two aspects are of concern:

1. How well the algorithm finds global minima, and
2. how quickly it manages to do so.

The best-achieved quality score is used to assess the first aspect of performance. Due to the black-box nature of the optimization and the simulator's stochasticity, determining

the global minimum is impossible. Therefore, only raw quality values are considered without making attempts to evaluate their proximity to the true global minimum.

To address the secondary concern, multiple performance metrics are observed. One such KPI is the number of simulations needed to arrive at the final result (sr_{final}). Furthermore, since the quality values of simulator runs with the same inputs vary slightly, a small improvement in the quality value could happen by pure chance. Hence, the number of simulation runs until the optimization enters a certain range around the final value is also observed and is referred to as the number of runs until convergence (sr_{conv}). For this range, the standard deviation of ± 0.0065, observed among the quality scores of simulation runs with equal parameters, is used. Furthermore, since all presented algorithms are capable of parallel execution during an iteration, one could argue that it may be more representative of the actual runtime to measure the number of iterations until the final value is discovered or the algorithm converges. Therefore, these performance indicators are also observed (it_{final}, it_{conv}). However, it should be noted that the degree of parallelism can easily exceed the number of available cores on the machine where the optimization is executed. Hence, the number of required iterations cannot be considered a strictly better performance indicator than the simulation runs, and vice versa. Additionally, a lower number of simulation runs can also indicate that the algorithm generated many invalid solutions for which the simulation was skipped. These metrics are therefore only considered secondary, and decisions are primarily based on the quality of the found solutions.

6.2.3 Hyperparameter Selection Strategy

For each algorithm, a limited set of parameters influences the maximum simulation runs per iteration. Hyperparameters outside this set are tuned based on the reduced-scale experiments. The remaining hyperparameters are then chosen in a way such that the theoretical limit of simulation runs for the scenario described in Section 6.1 is 5000 assuming 100 iterations. If an algorithm cannot be tuned this way, it is considered unfit for the given scenario, as running hyperparameter optimization on large-scale scenarios would be impractical.

6.2.4 Tuning Results

For each metaheuristic, the five best-performing parameter configurations with respect to result quality are analyzed concerning the presented optimizer KPIs. They are listed in Table 6.2. The table has been extended with several derived parameters that are calculated from a combination of two hyperparameters.

To uncover monotonic relationships between the tuned hyperparameters and the given KPIs, a correlation analysis is performed, the results of which are listed in Tables 6.3 and 6.5. To that end, the Spearman Rank Correlation is used, which measures monotonic relationships by comparing ranked lists of values, as opposed to uncovering linear relationships like Pearson's coefficient. This is done for two reasons. Firstly, it is

PSO

n	c1	c2	w	c1/c2	init rand	cons strat	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
25	1.39	1.80	0.42	0.77	False	$reject_{rel}$	0.260694	337.4	150.8	25.4	11.2
26	0.97	0.55	0.55	1.76	False	$reject_{max}$	0.263622	253.2	127.2	20.2	11.2
22	1.37	1.63	0.67	0.84	True	$reject_{rel}$	0.266682	62.4	58.4	16.8	16.2
24	0.98	1.95	0.46	0.50	True	$reject_{max}$	0.267177	103.2	70.2	22.0	16.2
21	1.73	1.23	0.75	1.41	False	$reject_{max}$	0.267643	97.6	81.4	16.6	12.4

GA

n	cp	mp	p	e	ss	p/n	ec/n	init rand	cons strat	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
44	0.60	0.13	9	1	rws	0.20	0.02	False	$reject_{max}$	0.264592	463.8	188.8	15.6	5.4
24	0.85	0.43	7	4	rws	0.29	0.17	False	$reject_{max}$	0.265035	200.8	114.6	23.8	11.8
49	0.38	0.69	6	2	sus	0.12	0.04	True	$reject_{max}$	0.265111	174.6	86.4	18.4	8.4
49	0.46	0.30	4	3	sus	0.08	0.06	False	$reject_{max}$	0.265410	437.8	164.0	18.4	5.8
32	0.86	0.19	7	3	tmnt	0.22	0.09	False	$reject_{rel}$	0.265429	286.6	191.6	18.4	11.0

ABC

w	o	l	w/o	w+o	init rand	cons strat	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
12	16	1	0.75	28	False	$reject_{max}$	0.261980	406.0	197.4	24.2	13.0
15	17	2	0.88	32	False	$reject_{max}$	0.262297	481.6	287.2	24.8	15.8
8	10	3	0.80	18	False	$reject_{rel}$	0.263523	272.4	207.4	28.2	22.6
20	17	2	1.18	37	False	$reject_{max}$	0.264176	461.4	285.2	22.8	14.8
8	13	4	0.65	21	True	$reject_{rel}$	0.265185	320.0	123.6	23.2	10.2

DE

n	cp	dw	init rand	cons strat	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
29	0.75	1.81	True	$reject_{rel}$	0.264531	272.0	208.4	25.2	21.0
16	0.90	0.53	False	$reject_{rel}$	0.264604	319.2	115.6	26.6	13.0
12	0.76	0.36	False	$reject_{max}$	0.264662	207.2	135.8	23.6	17.2
14	0.84	1.30	False	$reject_{rel}$	0.265519	157.2	45.2	26.8	13.0
26	0.96	1.34	True	$reject_{rel}$	0.265642	285.0	179.2	28.8	20.6

CSO

n	p_a	init rand	cons strat	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
17	0.70	Falsed	$reject_{max}$	0.272702	47.2	19.4	25.0	13.0
24	0.88	False	$reject_{max}$	0.274817	42.8	37.8	15.0	13.0
11	0.81	False	$reject_{max}$	0.277119	31.6	20.2	20.4	14.2
20	0.75	True	$reject_{max}$	0.279177	24.8	23.0	16.4	14.8
9	0.45	False	$reject_{max}$	0.280014	15.2	13.6	20.6	18.4

Table 6.2: Results of HPO experiments with best quality scores of each algorithm.

likely that certain hyperparameters influence the KPIs super- or sub-linearly. Secondly, while most hyperparameters are real-valued, the choices of random initialization and constraint handling strategy are binary and are therefore best analyzed using rank correlation. To calculate a coefficient for the choice of *random initialization*, the values have been transformed such that $False = 0$ and $True = 1$. Similarly, the choice of *constraint strategy* has been transformed so that $reject_{max} = 0$ and $reject_{rel} = 1$. The analysis also includes the aforementioned derived parameters. The coefficients with the highest magnitude, ignoring the sign, for each KPI are highlighted in bold. They indicate the hyperparameter with the highest impact on the respective performance metric.

To allow a similar analysis of the choice of selection strategy for the GA, Table 6.4 lists the KPIs gathered across runs that use the respective strategies.

Table 6.6 lists the choices for all hyperparameters that were decided on. The rationale for each of them is given next.

PSO					
Parameter	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
n	**−0.6674**	+0.4238	**+0.5151**	+0.0317	−0.0956
c1	−0.2129	+0.2482	+0.3159	−0.0230	+0.0822
c2	+0.0399	−0.1422	−0.1316	+0.0648	−0.0015
w	+0.1527	**−0.5320**	−0.4003	**−0.3577**	+0.1825
c1/c2	−0.1444	+0.1609	+0.2301	−0.2456	+0.0309
init rand	+0.3974	−0.5109	−0.3974	−0.0569	**+0.5310**
cons strat	−0.0885	−0.1947	−0.3009	−0.0798	−0.5499
GA					
Parameter	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
n	−0.4322	**+0.5904**	**+0.5279**	+0.0785	−0.3441
cp	−0.2873	+0.2730	+0.3204	+0.1274	+0.0704
mp	+0.3718	−0.3252	−0.3139	+0.1214	**+0.4442**
p	−0.1864	−0.0350	−0.0403	+0.1544	+0.0495
ec	+0.3023	−0.4041	−0.3825	−0.1592	+0.0842
p/n	+0.2150	−0.4872	−0.4647	+0.0068	+0.2484
ec/n	+0.3956	−0.5483	−0.5024	−0.1899	+0.1709
init rand	+0.3747	−0.5142	−0.4967	**−0.4366**	−0.1309
cons strat	**+0.4425**	−0.2655	−0.2832	−0.2394	−0.1772

Table 6.3: Spearman correlation coefficients between hyperparameters and optimizer KPIs.

PSO

For PSO, *swarmsize* is the only parameter that controls the theoretical maximum number of simulation runs. Therefore, it was set to 50 to achieve the target of 5000 runs. *Social Influence* appears to be mostly irrelevant for the given setting, with the highest magnitude among the gathered correlation coefficients being −0.1316 with sr_{conv}. It was therefore simply set to a similar value as the one from the best-performing trial run. Higher *Personal Influence* improved quality with a coefficient of −0.2129, but increased simulation runs to convergence with a coefficient of +0.3159. Observing the correlation values of the ratio between the influence parameters reveals that it appears to have a minimum to slight impact on all observed KPIs with the highest magnitude being −0.2456 concerning it_{final}. It was therefore also simply set to the value used by the best-performing run. A higher *Inertia* seems to offer a good trade-off between quality and simulation runs. However, increasing it too much risks generating a lot of particle positions that violate constraints. This could also explain why this parameter greatly decreases simulation runs, observable by the coefficient of −0.5320 with sr_{final}. It was ultimately set to a balanced value that is quite close to the settings of the top five runs. When looking at the constraint handling strategy, $reject_{rel}$ seems to be the better choice across all observed performance indicators and was therefore chosen. Random

selection strategy	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
rws	0.267704	194.52	107.28	19.56	11.56
sus	0.267843	207.84	101.44	18.56	9.36
rank	0.276517	37.57	31.50	12.67	9.50
tournament	0.267158	194.10	120.25	18.10	9.15

Table 6.4: Mean values of optimizer KPIs for different GA selection strategies.

initialization appears to have a detrimental effect on quality and iterations to convergence, with coefficients of +0.3974 and +0.5310. However, it significantly reduces the total number of simulation runs, observable by a coefficient of −0.5109 with sr_{final}. It was still decided not to initialize particles randomly despite the potential performance hit.

GA

For the GA, *Population Size* primarily drives the number of possible simulation runs and is also one of the main drivers of quality indicated by a correlation coefficient of −0.4322. However, the library that was used is implemented in an efficient way, skipping all quality function executions that are not strictly necessary. Combined with the choice of *Selection Strategy* and *Elitism Count*, and a preliminary trial run, it was determined that 120 individuals should, on average, lead to the desired 5000 maximum simulation runs. Higher *Crossover Probability* seems to benefit quality a bit, shown by a coefficient of −0.2873, while also increasing simulation runs, indicated by a coefficient of +0.3204 with respect to sr_{conv}. It was therefore set to a high 0.8. A higher *Mutation Probability* appears to negatively impact quality and iteration-based convergence, having coefficients of +0.3718 and +0.4442, respectively, while decreasing the number of simulation runs needed, indicated by a coefficient of −0.3252 with sr_{final}. It was therefore kept at a rather low 0.2. The correlations for the number of *Mating Parents* at first seem to show a slight improvement in quality with higher values, observable by a coefficient of −0.1864. However, when examining the ratio of mating parents compared to the total population, it becomes apparent that having too many hurts quality, as the derived parameter shows a coefficient of +0.2150. Hence, the number of *Mating Parents* was set to 3. The raw *Elitism Count* and the ratio of elitism compared to population size reveal that keeping a large number of well-performing individuals across generations is quite detrimental to achieved quality as they resulted in coefficients of +0.3023 and +0.3956, respectively. Therefore, it was decided to keep *Elitism Count* at 1. Observing Table 6.4, no *Selection Strategy* stands out as an obvious choice. However, rank-based selection performed the worst in terms of quality, only achieving an average quality score of 0.276517. Ultimately, the choice was made to use tournament selection. As with PSO, random initialization significantly decreases quality, indicated by a coefficient of +0.3747. It was therefore decided to go with the seeded initialization strategy. The choice of constraint handling strategy presents a trade-off, with $reject_{max}$ improving quality and $reject_{rel}$ reducing simulation runs. The quality improvement of $reject_{max}$ outweighs the downsides. Hence, it was chosen.

ABC					
Parameter	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
w	−0.1653	+0.3775	+0.3912	−0.2527	**−0.3362**
o	**−0.8311**	**+0.9465**	**+0.9031**	+0.4374	−0.0019
l	+0.0085	+0.0031	+0.0117	+0.0521	+0.0572
w/o	+0.6025	−0.6724	−0.6125	**−0.6285**	−0.2351
w+o	−0.7188	+0.9033	+0.8458	+0.2065	−0.1802
init rand	+0.4254	−0.2384	−0.2001	−0.4856	−0.2225
cons strat	+0.0708	−0.2478	−0.2479	+0.2746	+0.3281
DE					
Parameter	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
n	**−0.6143**	**+0.5859**	**+0.4847**	+0.3084	+0.1043
cp	+0.1407	−0.2202	−0.2318	+0.1664	+0.1949
dw	+0.0737	−0.3377	−0.1376	+0.1136	**+0.5122**
init rand	+0.2428	−0.1475	−0.2082	**−0.4684**	−0.2346
cons strat	−0.3399	+0.2005	+0.1133	+0.2092	+0.2357
CSO					
Parameter	*quality*	sr_{final}	sr_{conv}	it_{final}	it_{conv}
n	−0.2095	+0.3231	+0.4319	**−0.3456**	−0.1581
p_a	**−0.7171**	**+0.8660**	**+0.8036**	+0.1523	+0.0581
init rand	+0.2702	−0.2877	−0.0784	−0.0262	**+0.5417**
cons strat	+0.4506	−0.2905	−0.2303	+0.0502	+0.3012

Table 6.5: Spearman correlation coefficients between hyperparameters and optimizer KPIs continued.

Overall, the quality scores of the solutions found by the GA runs vary only slightly, which may indicate that, while the GA does offer a lot of hyperparameters, it is not necessary to tune them perfectly to achieve good results in the given setting.

ABC

For ABC, the combined number of *Workers* and *Onlookers* drives the maximum number of simulation runs. Hence, the total number was chosen to be 50. A higher number of *Onlookers* appears to be very favorable for the quality of the result, indicated by a correlation coefficient of −0.8311. *Workers* also contribute but are less important, as their correlation with quality is only −0.1653. Furthermore, the observed *Worker*-to-*Onlooker* ratio indicates that having slightly fewer *Workers* than *Onlookers* is generally favorable for quality. However, the number of *Onlooker* bees is almost perfectly linearly correlated with both simulation-run-based KPIs, indicating that striking a balance between quality and runtime may be difficult. Increasing *Onlookers* seems to not affect the iterations to convergence at all, or their relationship is not monotonic, since the respective coefficient is only −0.0019. Ultimately, a ratio of 22 *Workers* to 28 *Onlookers* was chosen. The

Algorithm	Parameter	Chosen Value
PSO	n	50
	c1	1.4
	c2	1.8
	w	0.5
	init rand	False
	cons strat	$reject_{rel}$
GA	n	120
	cp	0.8
	mp	0.2
	p	3
	ec	1
	ss	tnmt
	init rand	False
	cons strat	$reject_{max}$
ABC	w	22
	o	28
	l	2
	init rand	False
	cons strat	$reject_{rel}$
DE	n	50
	cp	0.7
	dw	1.5
	init rand	False
	cons strat	$reject_{rel}$
CSO	n	80
	p_a	0.6
	init rand	False
	cons strat	$reject_{max}$
NSGA-II	n	120
	cp	0.8
	mp	0.2
	p	3
	ss	tnmt (NSGA-II)
	init rand	False
	cons strat	$reject_{max}$

Table 6.6: Hyperparameters chosen for optimizer experiments.

Limit appears to have a minimal impact on all observed KPIs, with no coefficient even reaching a magnitude of 0.01. It was set to 2, based on the average across the top five runs. Regarding the constraint handling strategy, choosing $reject_{max}$ appears to have a clear positive impact on the required iterations, indicated by coefficients of +0.2746 concerning it_{final} and +0.3281 concerning it_{conv}. However, $reject_{rel}$ seems to reduce the number of simulation runs, observable by the correlation of −0.2479 with sr_{final}. Quality appears to be almost unaffected by the choice of constraint handling strategy, indicated by a coefficient of +0.0708. $reject_{rel}$ was ultimately chosen. Due to a high positive impact on result quality, indicated by a coefficient of +0.4254, the seeded initialization strategy was chosen.

DE

The parameter driving the number of potential sim executions is *Population Size*. It was chosen to be 50, mapping to exactly 5000 potential simulation runs. An increase in *Crossover Probability* only had a minor impact on quality, observable by a correlation coefficient of +0.1407, and increased simulation runs slightly, which is shown by a coefficient of −0.22002 concerning sr_{final}. It was therefore set to 0.7, which is slightly lower than the values present among the top five runs. Either *Differential Weight* is completely irrelevant to the quality of the found solution, or the relationship between it and the quality KPI is not monotonous, indicated by a coefficient of +0.0737. However,

increasing it did decrease the number of necessary simulation runs while simultaneously increasing the iterations to convergence, shown by coefficients of −0.3377 and +0.5122, respectively. It was therefore chosen to be 1.5 to strike a balance between quality and the number of simulation runs. Random initialization slightly hurts overall quality, indicated by a coefficient of +0.2428, but appears to decrease the necessary simulation runs only a bit, shown by a coefficient of −0.2082 with respect to sr_{conv}. It was once again decided to use the seeded initialization strategy. The choice of constraint handling strategy had an impact on quality, indicated by a coefficient of −0.3399, showing a clear preference for $reject_{rel}$. However, this once again comes with the trade-off of causing slightly more simulation runs and iterations until convergence, indicated by coefficients of +0.1133 and +0.2357, respectively. This trade-off was determined to be worth it and $reject_{rel}$ was chosen as the constraint handling strategy.

CSO

Looking at CSO, the number of potential simulation runs is determined by the number of *Nests* and the *Abandonment Rate*. An increased number of *Nests* had the expected effect of increasing quality at the cost of simulation runs, indicated by correlation coefficients of −0.2095 and +0.3231, respectively. However, the highest positive impact on quality, observable as a coefficient of −0.7171, was caused by a higher *Abandonment Rate*. It also presents a major trade-off with both simulation-run-based KPIs, showing coefficients of +0.8660 and +0.8036. As a high *Abandonment Rate* would essentially make the algorithm a pure random search, it was decided to fixate this value at 0.6 and adjust the number of *Nests* accordingly to reach the 5000 potential simulation runs. This is achieved by choosing 80 *Nests*. The chosen constraint handling strategy also had a considerable impact on quality, observable in a correlation of +0.4506. The preferred choice was $reject_{max}$, which is also used by the five best performing runs. Hence, it was chosen. Injecting a predefined solution into the initial nests had a slight positive effect on result quality and iterations to convergence, and a more pronounced negative effect on the total number of required simulation runs. Coefficients of +0.2702, +0.5417, and −0.2877 reflect this. The other KPIs were almost unaffected by it. It was decided to once again use the seeded initialization strategy over a fully random one.

NSGA-II

NSGA-II could not be analyzed in the same way due to it being a multi-objective metaheuristic. Its hyperparameters were chosen to mirror the GA's, except that elitism is not needed, and tournament selection is based on non-dominated sorting.

6.3 Results

This section presents the results of the described experiments. Each experiment was repeated 10 times for each algorithm using the same hyperparameters each time. A baseline was established by collecting quality metrics and the associated *qualtiy* score

Metric	PSO	GA	ABC	DE	CSO	NSGA-II
$quality_{mean}$	0.421403	0.427016	**0.414321**	0.421732	0.444638	N/A
$quality_{max}$	0.425312	0.429385	0.424934	**0.430649**	0.455557	N/A
$quality_{min}$	0.418205	0.423384	**0.365410**	0.394027	0.434141	N/A
$quality_{stdiv}$	0.002295	**0.002033**	0.016925	0.010337	0.007858	N/A
$sr_{final,mean}$	847.4	304.4	1345.7	232.5	**4.0**	277.1
$sr_{final,max}$	1994	568	2502	441	**13**	341
$sr_{final,min}$	224	159	183	102	**1**	216
$sr_{final,stdiv}$	588.4	111.4	785.5	109.5	**3.4**	39.4
$sr_{conv,mean}$	525.2	266.1	1204.9	207.4	**3.4**	N/A
$sr_{conv,max}$	1312	425	2502	441	**8**	N/A
$sr_{conv,min}$	118	159	183	102	**1**	N/A
$sr_{conv,stdiv}$	388.4	73.6	802.5	97.6	**2.2**	N/A
$it_{final,mean}$	73.4	41.3	56.1	63.1	**33.1**	93.1
$it_{final,max}$	99	97	98	98	**79**	98
$it_{final,min}$	33	**1**	7	38	**1**	71
$it_{final,stdiv}$	21.8	40.4	32.8	19.8	32.1	**8.0**
$it_{conv,mean}$	58.6	31.7	50.6	58.5	**26.8**	N/A
$it_{conv,max}$	84	91	98	97	**71**	N/A
$it_{conv,min}$	21	**1**	7	38	**1**	N/A
$it_{conv,stdiv}$	20.8	37.3	34.2	**17.3**	28.5	N/A

Table 6.7: Performance metrics of optimization algorithms obtained from experiments.

from 10 simulation runs using a set of parameters where p_{min} and p_{max} were set to 0.3 and 0.7 for each zone, respectively. This resulted in an average *quality* score of 0.4536 and a standard deviation of 0.0019, which is to be expected, as the simulator used is not fully deterministic and subject to small variations between runs. The baseline parameters were also used for seeded initialization.

6.3.1 Performance Results of Optimizers

This subsection presents an evaluation of the optimization algorithms with respect to the performance indicators introduced in Subsection 6.2.2. Additionally, the development of quality values throughout the optimization process is analyzed in relation to the iterations taken and the simulation runs performed for each algorithm. Finally, the algorithms are directly compared against each other to serve as motivation for a recommendation of an approach.

Analysis of Optimizer KPIs

Table 6.7 summarizes various statistics of optimizer KPIs gathered during the performed optimization runs. It shows the mean, minimum, maximum, and standard deviation for the quality score of the final result of an optimization run (*quality*), the simulation

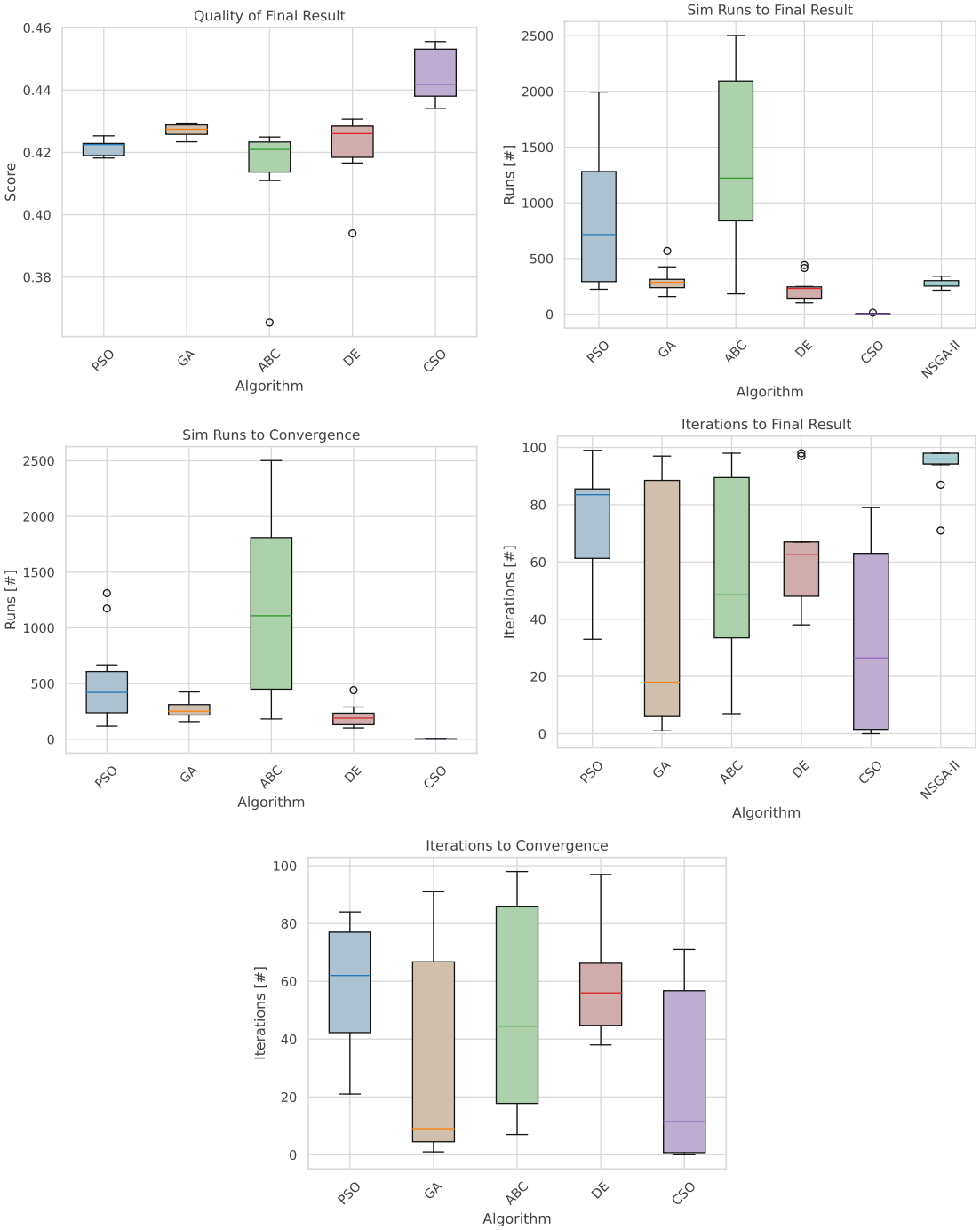

Figure 6.4: Performance metrics of optimization algorithms obtained from experiments visualized.

runs that were needed to arrive at the final result (sr_{final}), the simulation runs that were needed until the algorithm converged (sr_{conv}), and the respective metrics regarding iterations (it_{final}, it_{conv}). For all instances, a lower score can generally be considered better, as long as it does not come at the detriment of another KPI. Convergence is defined as a quality score difference of less than 0.0019 from the final result, based on the baseline's standard deviation between runs. Bold values indicate the lowest value observed in the row, and underlined values indicate the highest. The same performance indicators are visualized in Figure 6.4. Note that not all KPIs are applicable to NSGA-II, as it does not calculate a quality score during optimization.

With respect to the quality score, there are two noticeable anomalies among the observed data points. Both ABC and DE each had one optimization run, where they found a solution that is rated far better than any of the others, having associated quality scores of 0.365410 and 0.394027 respectively. All other optimizations that were conducted using these algorithms performed more in line with the results of the other algorithms. This hints towards the solution space of the problem being highly complex and most likely multi-modal. Further analysis will revisit these claims. Furthermore, CSO performed the worst in this regard, achieving an average quality score of 0.444638. In the best case, a solution that was slightly better than the baseline was found, but sometimes the final result would even be worse than the baseline. This can be explained by fluctuations in the simulator combined with the CSO's low simulation run count. PSO, ABC and DE, not considering the aforementioned abnormally low scores, performed quite similarly to one another, with ABC showing the lowest mean at 0.414321 and PSO having the most consistent quality score among them with an observed standard deviation of 0.002295. The GA was very consistent concerning the quality of the final result achieving the lowest observed standard deviation of 0.002033. However, the quality of the results was generally worse than that of the other three well-performing algorithms. While this may stem from a poorly configured algorithm, the consistent results observed during hyperparameter tuning indicate otherwise.

Compared to the baseline, PSO improved the aggregated quality score by 7.1 % on average and 7.8 % at best. GA achieved 5.9 % and 6.7 % improvements, respectively. ABC showed an 8.6 % average and a 19.4 % maximum improvement achieved by the outlier solution. DE showed a 7.0 % average and a 13.1 % maximum improvement achieved by the outlier solution. CSO only managed a 2.0 % average and 4.3 % maximum improvement.

Looking at the simulation runs needed to arrive at the final result, ABC required by far the most with an average of 1345.7 runs. The reason for this can be hypothesized to be the algorithmic design of ABC itself. More focus is put on refining already found solutions over finding new ones, which drives the optimization process into regions that do not violate constraints, hence not causing many skips of the simulation. The second most simulation runs were needed by PSO with an average of 847.4 runs. Also, here, the algorithmic property of particles gravitating towards non-constraint-violating regions can be listed as a potential reason for this behavior. The GA and DE had a similarly

low number of simulation runs, showing averages of 304.4 and 232.5 runs, respectively. Both algorithms rely on some form of random recombination and mutation of solution candidates to drive the optimization. This mechanism appears to be rather inefficient when faced with such a highly constrained search space. The simulation runs that the NSGA-II needed to arrive at a final Pareto front were very similar to the GA at 277.1 on average. This is to be expected, as both used the same configuration of hyperparameters. CSO by far executed the least simulation runs with the maximum runs being only 13. In some instances, CSO only executed a single simulation run for the injected starting value. The algorithm relies primarily on Lévy-flight-based random search, which appears to be highly inefficient at exploring valid areas of a highly constrained search space.

When comparing the simulation runs to convergence versus the final value, the algorithms' characteristics remain largely the same. However, PSO converges much faster than it reaches its final value, only needing 525.2 runs on average. This suggests that significant runtime is spent refining already good solutions.

Shifting the analysis over to iterations needed to arrive at a final value, PSO and NSGA-II found theirs rather late, at average iteration counts of 73.4 and 93.1 respectively. The GA, in general, finds the final value the fastest among decently performing approaches with an average iteration count of 41.3. However, it is also likely to take a lot longer in rare instances indicated by a large standard deviation of 40.4. There is one GA run that found the final configuration in the very first iteration among the starting population. This behavior is explainable by the fact that the GA generates a starting population of only valid solutions, leading to 120 simulation runs always happening in the first iteration. It can therefore be considered a rather front-heavy algorithm. CSO also showed a rather large spread of this performance indicator with a standard deviation of 32.1. However, analyzing CSO regarding this quality is rather pointless considering the low number of simulation runs that it conducts during the optimization process. ABC and DE present themselves as the most balanced algorithms with respect to this KPI, taking on average 56.1 and 63.1 iterations. However, DE's results regarding this KPI are more consistent, showing a standard deviation of 19.8 compared to 32.8 for ABC.

When comparing iterations for convergence versus final value, the trends are similar to simulation-run-based analysis. PSO requires noticeably fewer iterations to converge, with 58.6 iterations on average. Although this is a noticeable decrease, this still represents the highest value observed among optimizers. GA iterations also decrease, coming in at an average of 31.7, but a high spread persists indicated by a standard deviation of 37.3, the highest value observed.

Analysis of Quality Improvements by Simulation Runs

Figure 6.5 visualizes how the quality value of the current best-found solution develops throughout the algorithms' runtime concerning the number of simulation runs that were executed. Note that the graphs are intended to highlight trends and distributions of quality scores and therefore do not all use the same scales across both axes. They

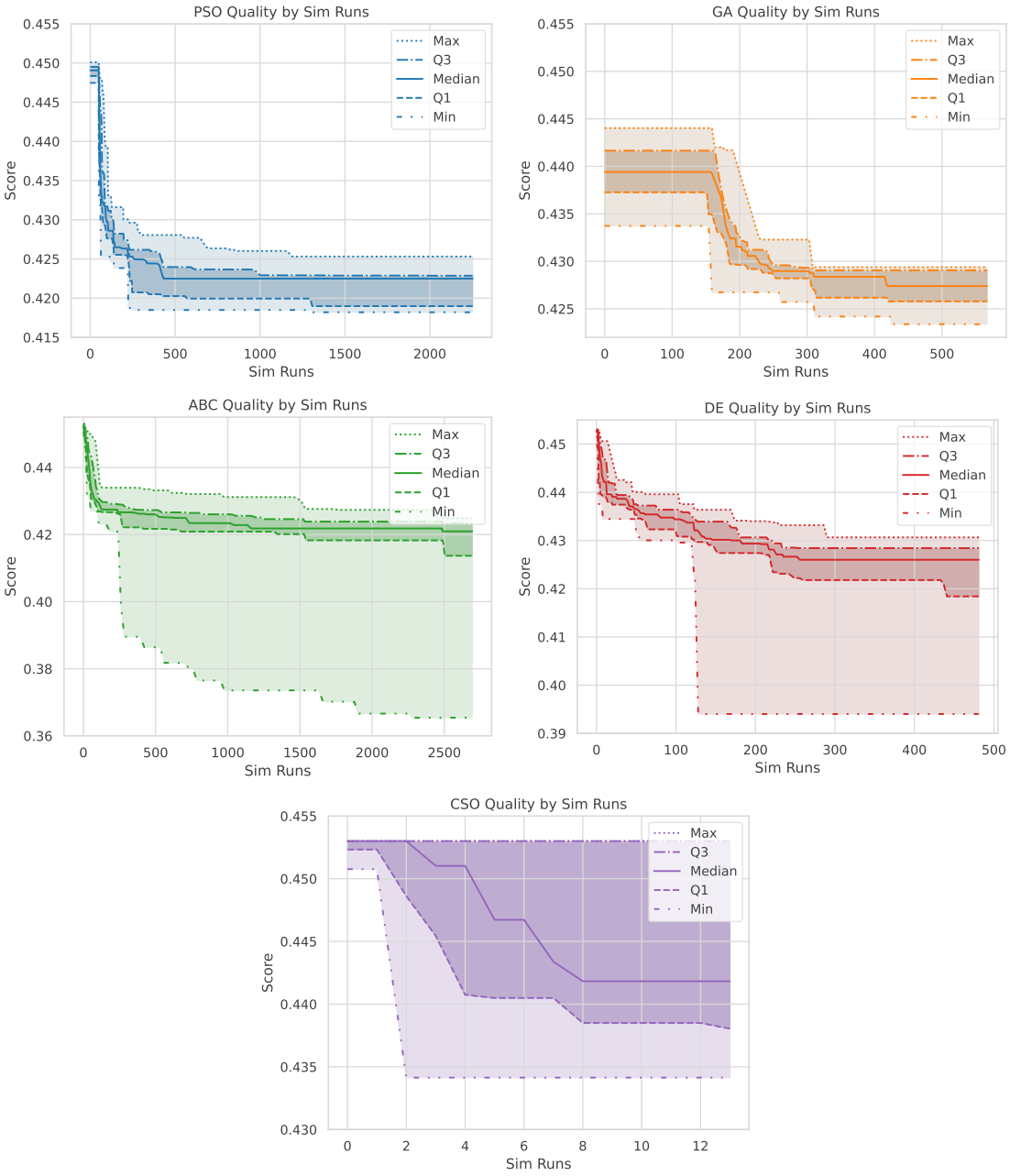

Figure 6.5: Development of quality values during optimization with respect to simulation runs.

essentially show a boxplot diagram at each point of the optimization process. Since simulation runs happen batched, missing values were interpolated between measured data points. Values above the measured baseline quality were disregarded. However, this only has an effect on the graph for CSO. Once again, since these visualizations use the quality score, NSGA-II is excluded from the analysis.

As already seen in the discussed statistics, the PSO quickly converges to a relatively good solution which is only improved by a small amount after around 500 simulation runs. The spread of the final quality values is low.

Looking at the GA confirms the assumption that its first iteration runs a large batch of simulations and after that the number of simulations per iteration drastically decreases, due to crossover and mutation producing invalid solutions. However, the GA is still able to tune the parameters after that initial iteration. The quality values of the results are very consistent.

The ABC shows an interesting pattern, as all quartiles and the maximum follow a characteristic curve like the other algorithms. However, during one run, a massive leap in quality occurred after around 250 simulation runs, resulting in the extremely low quality score of that run. This outlier was still improved after the quality jump.

A similar situation is presented in the graph for DE. Also, here, a massive leap in quality occurred for one run after around 110 simulation runs. The resulting parameters were also not further refined afterward, indicating more of a random fluke than a capability of the algorithm. Other than that outlier, DE also follows a curve similar to the other algorithms, albeit with a larger spread among the final quality scores.

The graph for CSO highlights the already discussed characteristics that identify the algorithm as unsuitable for the given scenario. 25 % of all runs did not even produce a value that was better than the baseline. Improvements in quality appear to be almost entirely random.

Analysis of Quality Improvements by Iterations

Figure 6.6 shows the development of the quality score of the current best solution over iterations of the algorithms. Once again, the y-axis has been scaled differently between graphs to emphasize interesting characteristics for each algorithm individually.

The PSO iteration curve largely mirrors its simulation-run-based counterpart. However, it reveals that PSO can still improve solutions in later iterations. It seems that many simulation runs might be skipped early on, while more are executed later, potentially explaining the flat tail in the simulation run curve versus the less flat iteration curve.

The GA plot once again shows the characteristic behavior of achieving the largest leap in quality in the first iteration. All curves, except for the one of the maximum, flatten out quite quickly, indicating that the GA would probably achieve similar results when using fewer iterations.

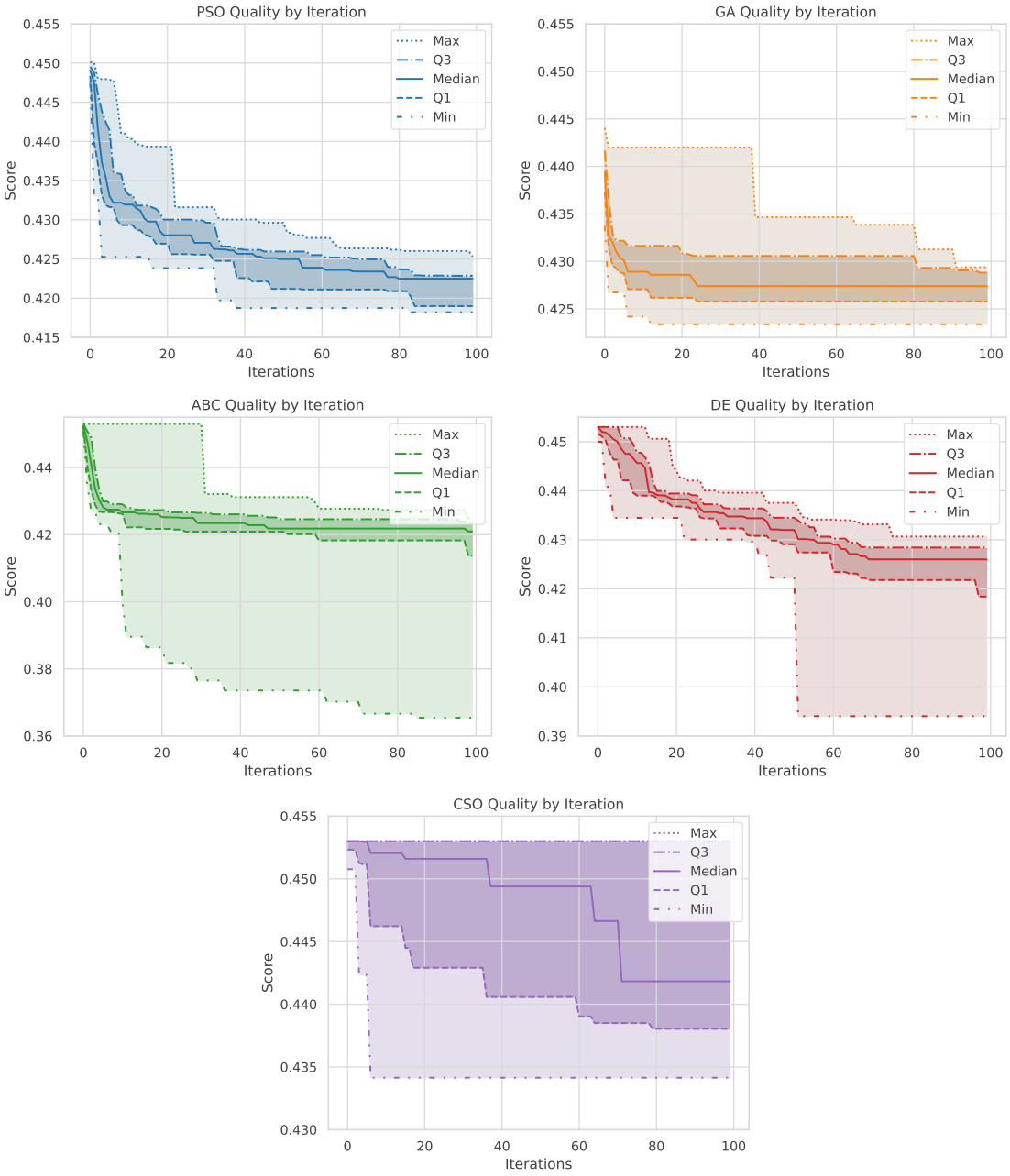

Figure 6.6: Development of quality values during optimization with respect to iterations.

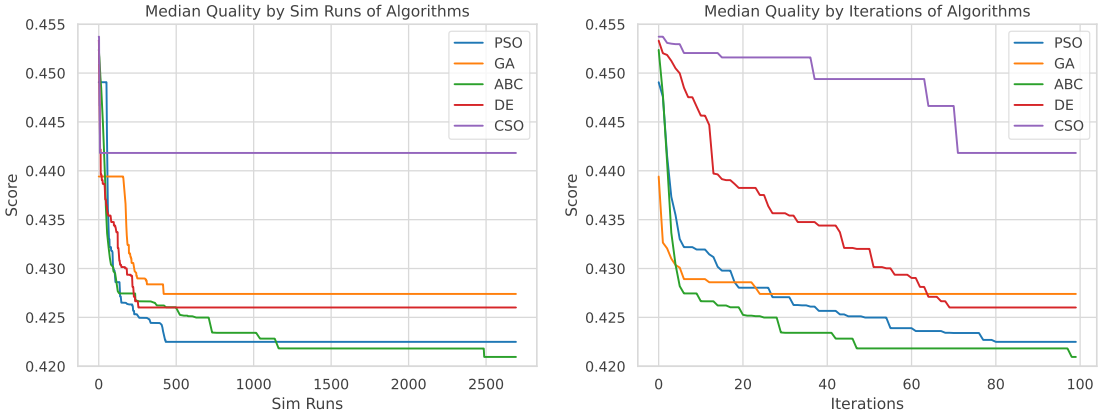

Figure 6.7: Comparison of algorithms regarding development of quality values.

The ABC plot again shows a large disparity between the curve of the best optimization run and that of all other curves. A difference to the simulation-run-based counterpart is presented by the curve of the worst run. It appears that for many iterations, the algorithm did not execute many simulation runs. However, the curve eventually catches up with the others.

Also, for DE, the curve is mostly similar to the simulation-run-based counterpart. However, one interesting aspect is that the outlier run seems to have made its leap quite late in the optimization process, and the bulk of simulation runs happened after. This may hint towards the algorithm searching in invalid sections of the search space for an extended period before making the lucky discovery of the well-performing parameter set.

The CSO curve matches its simulation-run-based counterpart for the most part in that it almost resembles a rectangle between a lot of very bad runs and one decent one. The best run found its parameters quite early on. It appears that improvements happened during the entire optimization process, leading to a more linear curve than the other algorithms show. This once again hints towards CSO being more of a random search scheme than a targeted optimization process.

Optimizer Comparison

To compare the algorithms, all median curves of the quality development regarding simulation runs and iterations are combined in Figure 6.7. CSO performs worst across both categories. The GA shows very fast convergence with respect to both viewpoints and comes in second to last place when it comes to the final quality score. Regarding simulation runs, DE is constantly slightly ahead of the GA. However, when viewed with respect to iterations, DE lags behind the GA for the majority of the optimization process and only catches up shortly before it converges. PSO is the second-best metaheuristic regarding quality improvements per iteration, always staying behind ABC. However, when viewing from a simulation-run-based perspective, PSO is ahead for the first 1000

simulation runs, where it is then overtaken by ABC. However, PSO also converges far earlier at around 500 simulation runs. ABC shows improvement as late as after 2500 simulation runs, making these bumps in quality the most expensive regarding computation effort compared to all other algorithms.

6.3.2 Performance Results of Tuned Autoscaling Configurations

This subsection analyses the configurations returned by the optimization algorithms with respect to the eight KPIs they were optimized for. Analysis is performed from two viewpoints. First, all discovered configurations per algorithm are analyzed together to discover average trends. Subsequently, only the best-performing parameters discovered by each algorithm are analyzed in the same way.

Analysis of Average KPI Results

Each configuration resulting from optimization was used in a simulation 10 times, and performance KPIs were extracted. Hence, for each of the five single-objective algorithms, 10 parameter sets were evaluated. For NSGA-II, all members of the final Pareto front were included. Figures 6.8 and 6.9 visualize the results aggregated over all such simulation runs for a given algorithm. For NSGA-II, the following approach was followed. For each Pareto front resulting from an experiment, two sets of parameters were chosen. The one that favored the analyzed KPI the most and the one that favored it the least. These categories are visualized separately. However, since NSGA-II usually produced rather small Pareto fronts, there is at most a minimal difference between categories.

All algorithms managed to reduce the average number of RTT-based SLO violations compared to the baseline. The biggest improvements were made by ABC, PSO and DE, which improved RTT-SLO-rate by 5.5 %, 4.8 % and 4.3 % when comparing medians with the baseline. CSO reduced this metric the least, shortly followed by both categories of NSGA-II-optimized parameters. The overall range of RTT-SLO-rate was quite wide for each algorithm, indicating that this may be a rather volatile target metric.

Cloud processing rate also mostly decreased across the board, except DE being the only algorithm that on average favored raising the CvE-rate, increasing it slightly by 0.7 %. NSGA-II kept it around the same level as the baseline. The configurations resulting from ABC optimization favored this metric the most followed by GA and PSO reducing it by 5.3 %, 4.0 % and 2.6 % respectively. Overall, the median difference in CvE-rate between optimizers is quite small, being less than 1% in all cases.

The number of zone-crossing requests increased on average for all algorithms compared to the baseline, suggesting that this metric was sacrificed to improve others. This aligns with its lower weight in the quality score calculation. However, the same patterns can be observed for NSGA-II, which uses no weights. Hence, it can be hypothesized that the chosen weight for this parameter in the quality quality function was a well-founded and reasonable choice. In general, DE favored this metric the least, increasing it by 3.0 %

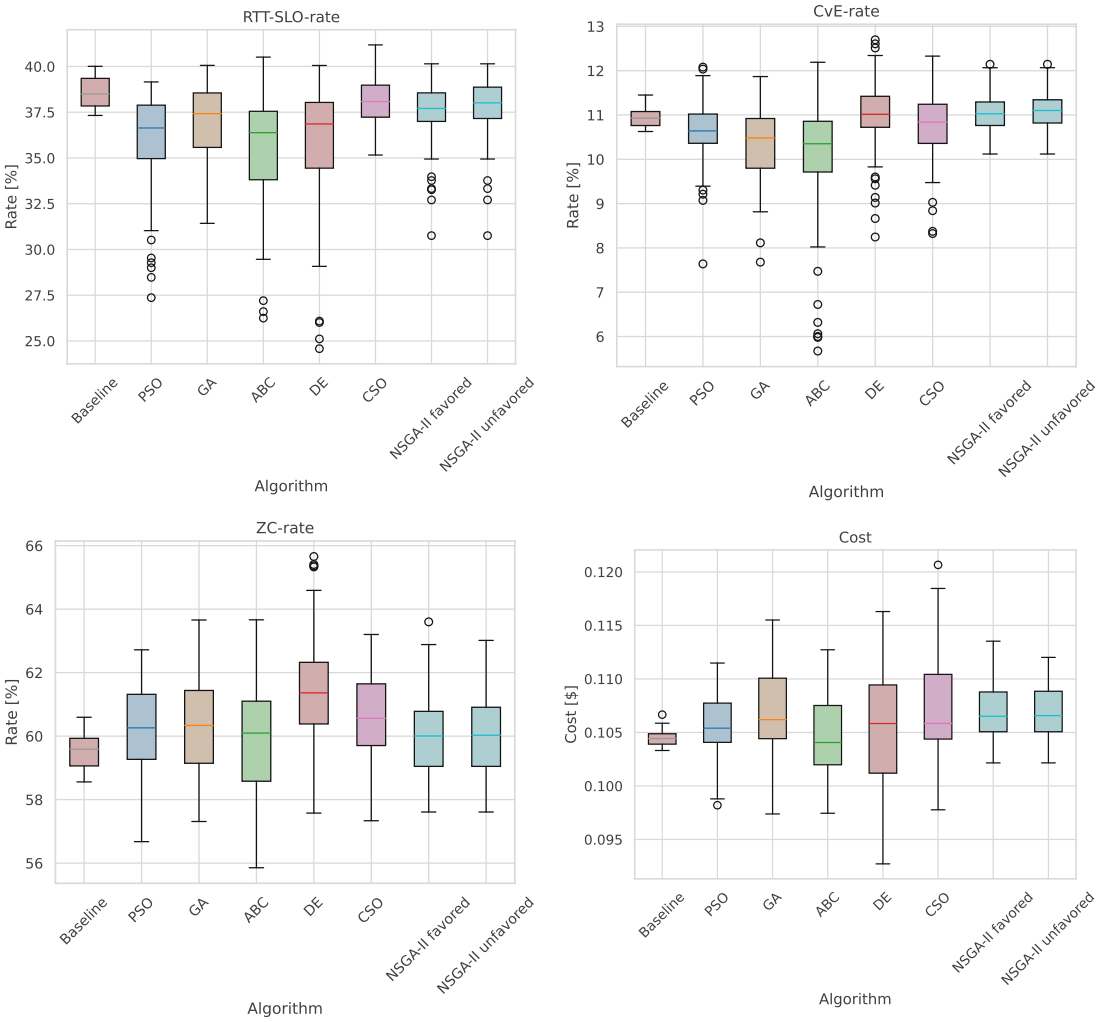

Figure 6.8: Target KPIs aggregated over all configurations resulting from optimization experiments.

followed by CSO, which increased it by 1.6 %. The other algorithms show roughly equal medians and ranges.

Cost was also slightly increased across all algorithms with respect to the baseline, except ABC, which on average decreased the operational cost by 0.4 %. No extreme outliers regarding extraordinarily low or high cost can be observed, as the difference between algorithms is less than 1 cent across all medians. Cost appears to be a stable metric.

CPU-SLO-max was very consistent, with most data points at the maximum value. This is expected given the CPU-intensive workload and high load of 80 % of the theoretical capacity. Interestingly, some GA, ABC, and DE parameter sets lowered this metric, even below 50 %, but these appear to be statistical outliers.

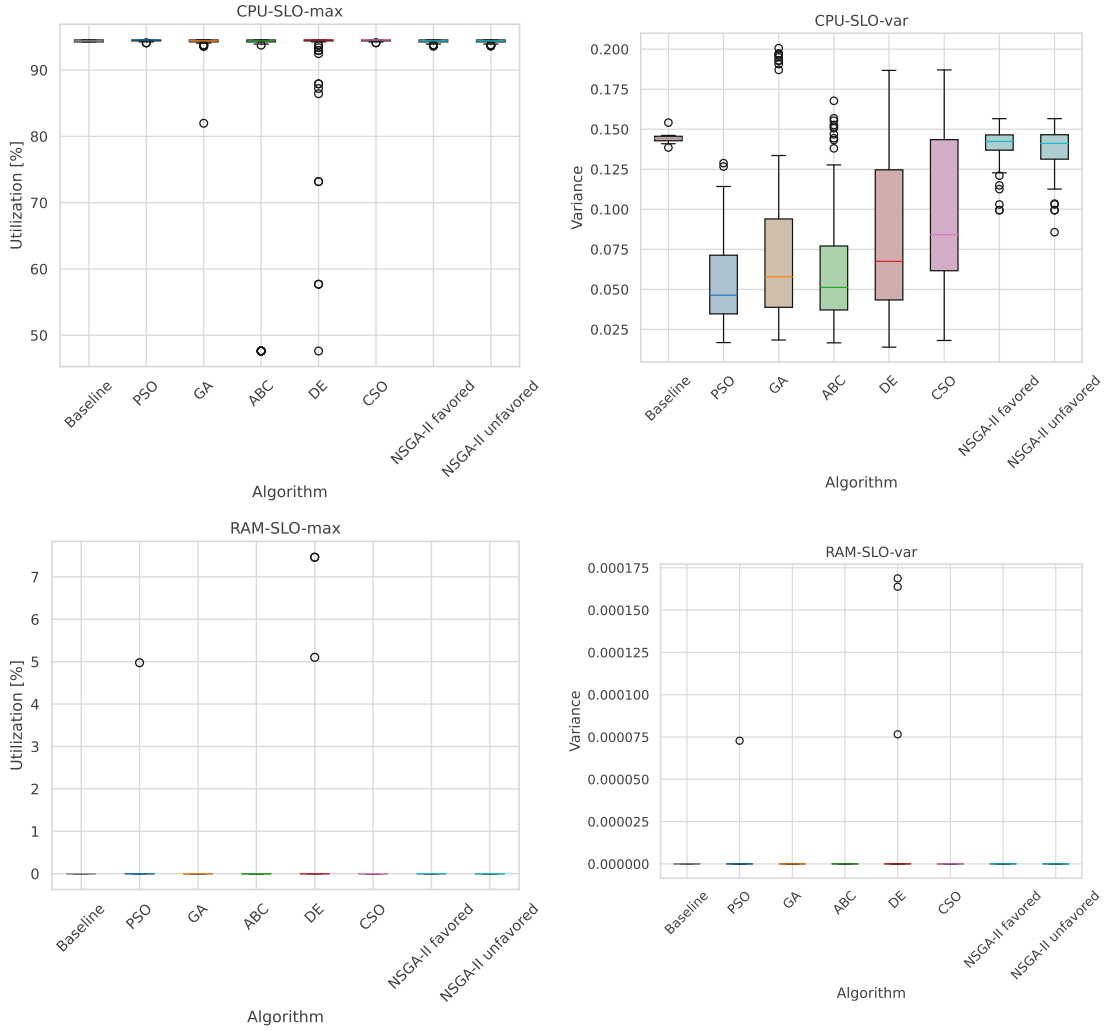

Figure 6.9: Target KPIs aggregated over all configurations resulting from optimization experiments continued.

CPU-SLO-var is the most diverse among the observed metrics. Every algorithm on average lowered it with respect to the baseline, meaning that the optimized configurations caused a more even distribution of load among the system. PSO, ABC and the GA all greatly favored this metric, reducing it by 67.9 %, 59.9 % and 64.4 %, respectively. DE and CSO both showed a rather high variance among their resulting parameters. Interestingly, NSGA-II failed to find a solution that significantly favored this metric, despite there being parameter configurations that were capable of doing so. This suggests that the high dimensionality of both search and objective space may hold back NSGA-II from finding representative Pareto fronts.

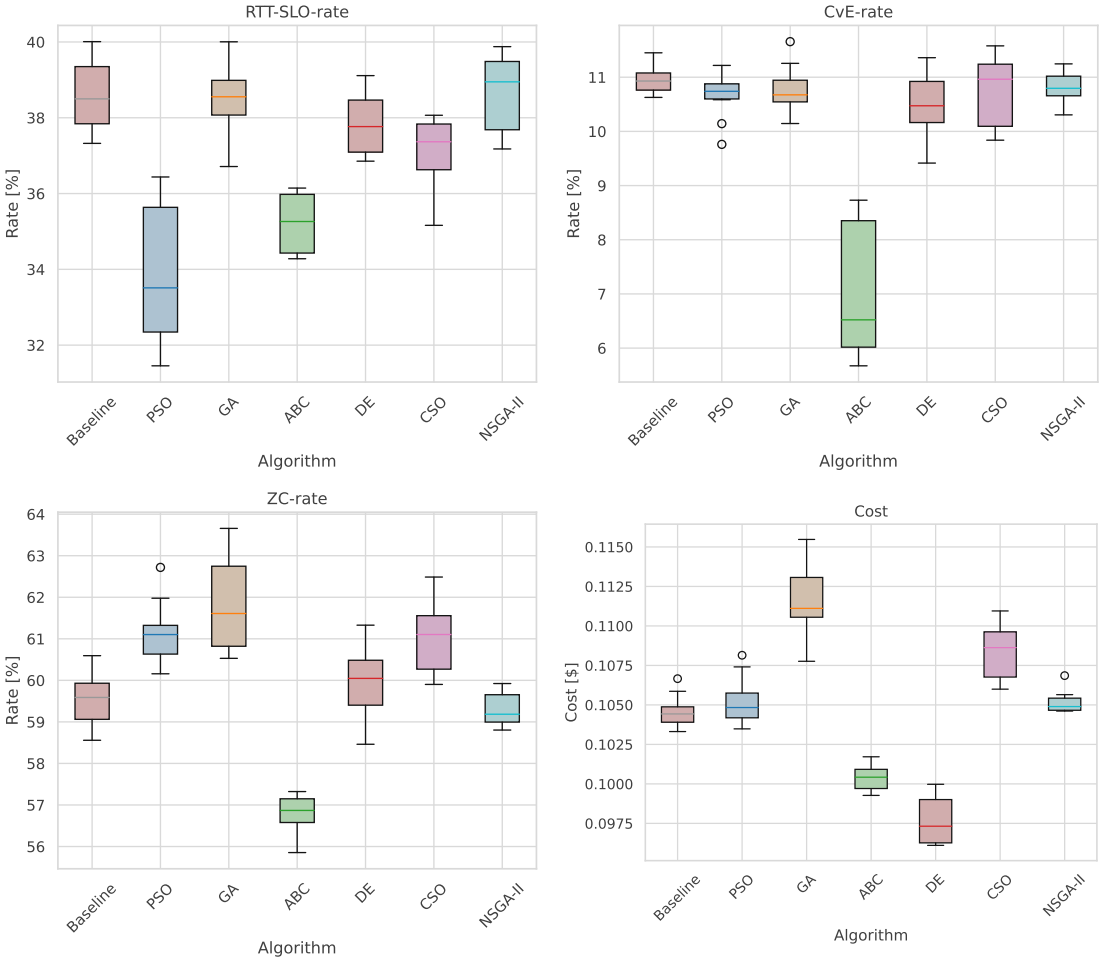

Figure 6.10: Target KPIs of the best-performing configurations resulting from optimization experiments.

Both RAM-based metrics were almost always 0, which is to be expected, as the function used in the experiments was not very memory intensive, making a RAM SLO violation a rare occurrence. There are only a few outlier data points that slightly raise either metric and only by negligible amounts.

Analysis of Best-Case KPI Results

The same visualizations that were presented above are provided in Figures 6.10 and 6.11. However, only simulation results that were obtained using the best parameter configuration with respect to the quality score returned by each algorithm are taken into consideration. This set also includes the two outlier solutions discovered by ABC and DE. For NSGA-II, all members of each Pareto front were combined and the one solution that showed the lowest value for the respective metric was chosen.

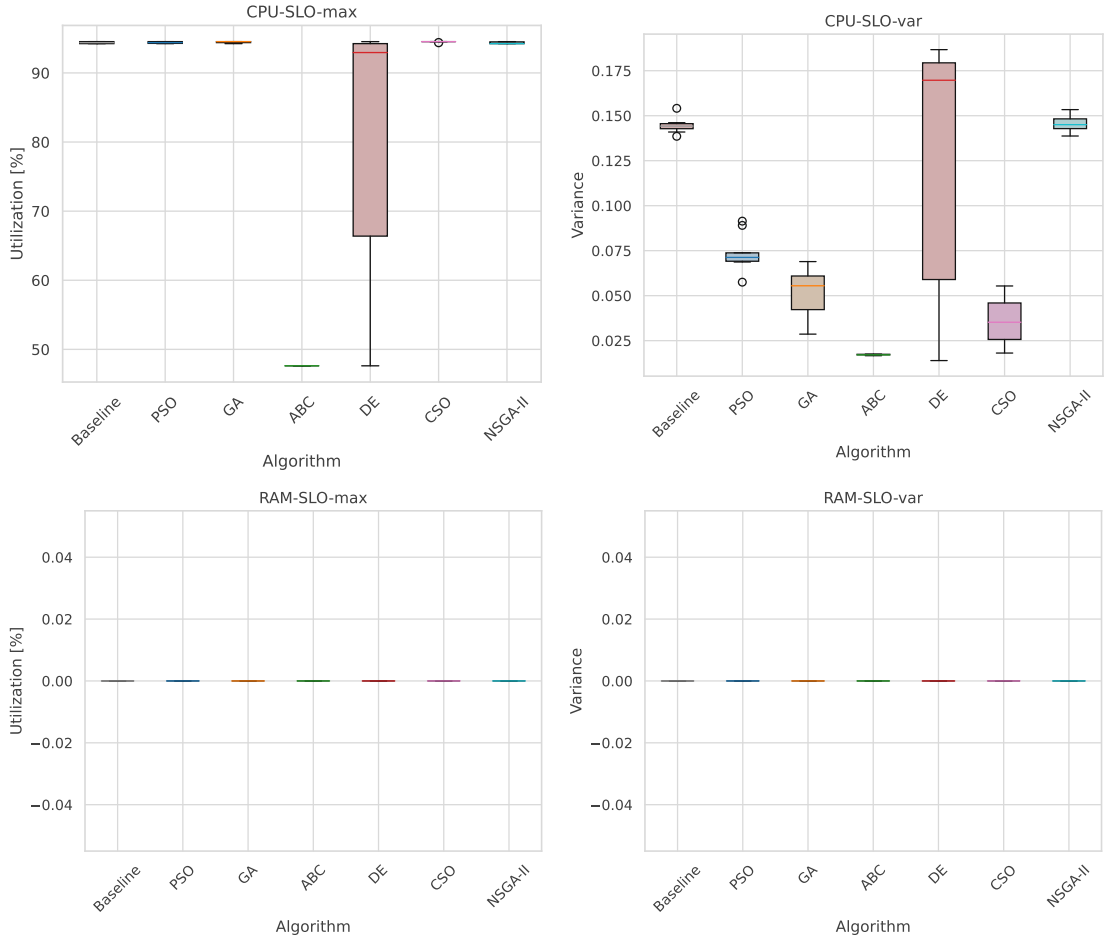

Figure 6.11: Target KPIs of the best-performing configurations resulting from optimization experiments continued.

The solution found by PSO was the one that reduced RTT-SLO-rate the most. It managed to achieve a 13.0 % improvement over the baseline when considering the medians, lowering the rate by an absolute 5.0 %. GA and NSGA-II both did not improve the number of SLO violations. The ABC outlier also decreased the rate by 8.4 %. However, the DE outlier only achieved a 1.9 % improvement over the baseline.

The boxplots for CvE-rate show a clear difference between the parameter sets. ABC managed to reduce this metric by 40.3 % compared to the baseline, lowering it by a rate of 4.0 %. The other algorithms only managed to reduce it slightly below the baseline. The DE-produced outlier did not noticeably boost this metric, only reducing it by 4.2 %.

A similar picture is painted by the results regarding ZC-rate. While most algorithms increased this KPI slightly, the outlier produced by ABC lowered it by 4.0 % compared to the baseline amounting to an absolute 3.0 %. The result discovered by NSGA-II also

Zone	PSO		GA		ABC		DE		CSO	
	p_{min}	p_{max}	p_{min}	p_{max}	p_{min}	p_{max}	p_{min}	p_{max}	p_{min}	p_{max}
IoT-Box-1	0.2917	0.6580	0.3000	0.8935	0.8559	0.9000	0.0035	0.7384	0.4434	0.4981
IoT-Box-2	0.2003	0.6499	0.2917	0.7928	0.1169	0.3909	0.1555	0.7176	0.3560	0.6325
IoT-Box-3	0.3063	0.6349	0.2307	0.6944	0.1000	0.7988	0.1874	0.5185	0.7631	0.8323
IoT-Box-4	0.2894	0.6806	0.3164	0.7559	0.1047	0.6255	0.3537	0.4815	0.8250	0.9469
Smartlet-5	0.3288	0.3654	0.2810	0.5553	0.1000	0.5298	0.2939	0.6332	0.6190	0.8415
Smartlet-6	0.2817	0.4800	0.2064	0.6269	0.1411	0.4432	0.1625	0.5565	0.4284	0.9124
Smartlet-7	0.0974	0.3445	0.2109	0.7174	0.1000	0.3224	0.1289	0.6016	0.4732	0.6557
Smartlet-8	0.2599	0.3997	0.7457	0.9996	0.1510	0.3041	0.1128	0.8243	0.5812	0.8193
Cloud	0.5726	0.9146	0.8274	0.9305	0.1661	0.2145	0.2072	0.3677	0.8534	0.8615
quality	0.418205		0.423384		**0.365410**		0.394027		0.434141	

Table 6.8: Best performing set of discovered pressure parameters for each algorithm.

did not increase it, instead keeping it around the same value as the baseline. Of the other algorithms, the GA increased it the most, while DE increased it the least. Although the differences between the algorithms are more noticeable here, the absolute changes in values are comparable to those of CvE-rate, the other processing location-based metric.

Cost follows a similar pattern as ZC-rate. However, the DE-discovered solution also significantly lowered this KPI by 6.8 %, even more so than the one produced by ABC, which lowered it by 3.8 %. This also marks the only KPI where the DE-discovered configuration managed to outperform the ABC-discovered one. The GA solution performs the worst, followed by the one produced by CSO. The PSO-produced solution slightly raised the cost above the baseline, and so did the one discovered by NSGA-II.

CPU-SLO-max shows an interesting result. Only the two outlier solutions managed to lower this metric below the maximum value. The solution obtained via ABC even managed to consistently keep the maximum rate below 50 %. The DE solution's ability to do so seems to be impacted greatly by random fluctuations during the simulation. However, it was also able to achieve a CPU-SLO-Max of 50 % during its best run. There is a high likelihood that DE discovered its solution when the given configuration produced the rare case, where it randomly performed very well regarding this metric. This would also explain why there was no future improvement after the initial discovery.

The trend of the DE parameter set being noticeably unstable continues when looking at CPU-SLO-var. Some of the runs performed with this parameter set even pushed the metric above the baseline. However, the ABC outlier solution managed to keep the load inequality consistently low. The solutions obtained from NSGA-II are comparable with the baseline. All other algorithms performed only slightly worse than ABC.

None of the best-performing parameter sets raised the RAM-based metrics above 0.

6.3.3 Analysis of Discovered Thresholds

Table 6.8 lists the parameter set discovered by each algorithm, which performed best with respect to its quality score. NSGA-II is excluded, as a definitive best-performing

parameter set cannot be defined. Looking at the values makes it clear that the discovered parameter sets are quite diverse. This once again hints towards the solution space being multi-modal and difficult to optimize. One pattern observed among the two outlier solutions discovered by ABC and DE is that the lower pressure thresholds in these solutions tend to be set at lower values. The ABC solution has eight lower pressure thresholds below 0.2 and the DE solution has six, compared to the PSO solution, which only has one pressure threshold set that low. Another pattern that can be observed is that the ABC- and DE-discovered solutions set both pressure thresholds for the cloud zone quite low. This means they prioritize scaling the cloud up quickly, which appears to have had a very positive impact on the overall system quality. While it may be true that parameters with these patterns perform better in the given scenario, it is also plausible that the areas around those solutions do not perform as well, since both these outlier solutions were only discovered once by their respective algorithms, and in both cases, the improvement came with one noticeable large leap instead of gradual refinement. In any case, it is shown that despite all of the algorithms, except CSO, being able to find autoscaler configurations that perform better than the baseline, none of them managed to consistently find configurations with an associated quality score below 0.4, of which the experiments prove that at least two exist, one of which cannot be attributed to the nondeterminism present in the simulator alone.

6.3.4 Analysis of Single- vs. Multi-Objective Optimization

A key concern with combining metrics into a single quality score is the potential for inaccurate representation due to manually chosen weights. While multi-objective optimization is a theoretical solution to this, the experiments suggest that this may not be the case for the presented scenario. Across all metrics used to arrive at the quality score, except for ZC-rate, NSGA-II failed to find solutions that performed better compared to those discovered by the single-objective algorithms. This does not prove that the presented *quality* function is a true lossless representation of system quality. However, it does show that in the given scenario, which is characterized by a highly constrained search space and a large number of objectives, single-objective approaches outperformed the evaluated multi-objective approach. A possible reason for this is that the NSGA-II struggled to find a representative Pareto front. Seven out of the 10 NSGA-II runs resulted in a Pareto front with a single member, two runs had two, and a single run resulted in three separate solutions. This practically eliminates the benefits of multi-objective optimization. It is possible that other multi-objective optimization schemes, which were not explored here, may perform better. Furthermore, it is also possible that the NSGA-II simply performed poorly due to being run with the hyperparameters discovered for the single-objective GA. However, since HPO becomes increasingly difficult for multi-objective algorithms, this is another strong argument against using NSGA-II in the given or a comparable setting. Finally, NSGA-II is likely to perform better with a reduced set of target KPIs. However, a large number of competing goals is a key characteristic of the given setting.

6.3.5 Key Observations

Distilling the experimental results down to the most important information derivable from them, we arrive at the following list of key takeaways.

- All algorithms, except CSO, consistently achieved improvements in quality score compared to the baseline.
- CSO is entirely unfit as an optimization algorithm for the given settings. This is likely due to the problem space being heavily constrained.
- NSGA-II, as a representative of muli-objective optimization algorithms, underperformed, when compared to the single-objective metaheuristics, which used an aggregate quality function.
- There are points in the problem space that have a comparatively good quality score associated with them. However, none of the observed algorithms managed to consistently discover them.
- ABC performed best in terms of the quality of found solutions. However, this performance comes at the cost of a high number of simulation runs, resulting in heavy computational effort. Additionally, ABC results are not the most consistent.
- DE also managed to find an outlier solution. However, the fact that, in the respective quality curve, there is a huge leap in the score and a flat tail afterwards indicates that this discovery most likely happened due to pure chance.
- The slight fluctuation between simulator runs is mostly unproblematic. However, in certain cases, as with the DE-discovered outlier, it did lead to problematic results.
- The biggest bumps in overall quality were observed among solutions that significantly lowered the resource-based metrics. The low variance between the values of these KPIs indicates that they may not be an ideal choice for the task at hand.
- PSO performed best in terms of efficiency regarding the number of simulation runs. It is only barely beaten by ABC after a large number of simulation runs. Additionally, PSO results are rather consistent.
- The fact that outlier solutions exist motivates running multiple optimizations for a given setting to improve the chances of discovering them.

CHAPTER 7

Robustness Analysis of Optimized Autoscaler Configurations

The goal of this chapter is to evaluate whether a static autoscaler configuration, that resulted from an optimization process, still performs well when the actual scenario differs from the one the parameters were optimized for. This is intended as a viability study of the presented approach and an experimental exploration to find the aspects of the optimization setting that are particularly critical. To this end, three scenarios are described in which different aspects of the setup are altered. These include the utilized infrastructure, the request patterns generated at each edge zone, and the overall load on the system. Experiments using them are run, and the results are subsequently analyzed. Each experiment compares the performance of a set of baseline parameters, where all lower pressure thresholds are set to 0.3 and all upper pressure thresholds are set to 0.7, with the best-performing parameter sets of PSO and ABC, which were discovered during the experiments described in Chapter 6. The PSO-discovered parameters were chosen because they resulted from the most efficient and consistent optimization scheme. The ABC-discovered parameters were chosen because they achieved the best quality score among all observed parameter sets. This way, in addition to the goals mentioned above, the experiments are used to evaluate if and why one of those two configurations is particularly robust or volatile regarding changes in the setting.

7.1 Differences in Infrastructure

This experiment tests the performance of an edge-cloud platform using an autoscaler configuration optimized for a slightly different infrastructure.

KPI	Baseline		PSO		ABC	
	absolute	relative	absolute	relative	absolute	relative
RTT-SLO-rate	**−0.112943**	**−29.3371 %**	−0.075200	−22.4403 %	−0.048709	−13.8133 %
CvE-rate	−0.004076	−3.7295 %	**−0.028430**	**−26.4711 %**	+0.013387	+20.5279 %
ZC-rate	+0.005079	+0.8524 %	**−0.029976**	**−4.9058 %**	+0.015855	+2.7881 %
Cost	−0.012331	**−11.8082 %**	−0.010446	−9.9645 %	+0.003092	+3.0786 %
CPU-SLO-max	+0.001433	+0.1518 %	**±0.000000**	**±0.0000 %**	+0.468663	+98.4192 %
CPU-SLO-var	**+0.003138**	**+2.1758 %**	+0.022573	+31.6795 %	+0.196624	+1150.2514 %

Table 7.1: Changes in KPIs of setting with altered infrastructure compared to the optimized setting.

7.1.1 Experimental Setup

The infrastructure used during the experimental simulations differs from the one described in Section 6.1 in the following way:

- Zone 1 was changed to a *Smartlet* zone,
- zone 4 was changed to a *Smartlet* zone, and
- zone 8 was changed to an *IoT-Box* zone.

Overall, this increases the total number of *Smartlet* zones and decreases the total number of *IoT-Box* zones by one each. This slightly increases the total platform capacity with respect to compute and memory resources.

The simulated workload is the same as the one described in Section 6.1 when only considering zone IDs for mapping zones to request profiles. The function that is being called is also the same. Theoretically, this, combined with the slight increase in total platform resources, should enable the platform to operate better than when using the original infrastructure.

A simulation with the given setup was executed 10 times for each parameter set to compensate for the simulator's slight nondeterminism.

7.1.2 Results

Figures 7.1 and 7.2 show how the KPIs change for each parameter set when running simulations with the altered infrastructure compared to the setting used to discover the configurations. Furthermore, Table 7.1 lists the absolute and relative differences of each KPI for each configuration. For each KPI, the absolute and relative changes that represent either the best improvement of the metric or the smallest degradation, if no improvement was observed, are highlighted in bold. Analogously underlined values represent the biggest degradation or the smallest improvement if none occurred. As a basis for calculating the differences, the median value of each KPI across all simulation runs was used. RAM-based metrics are omitted because they were consistently zero.

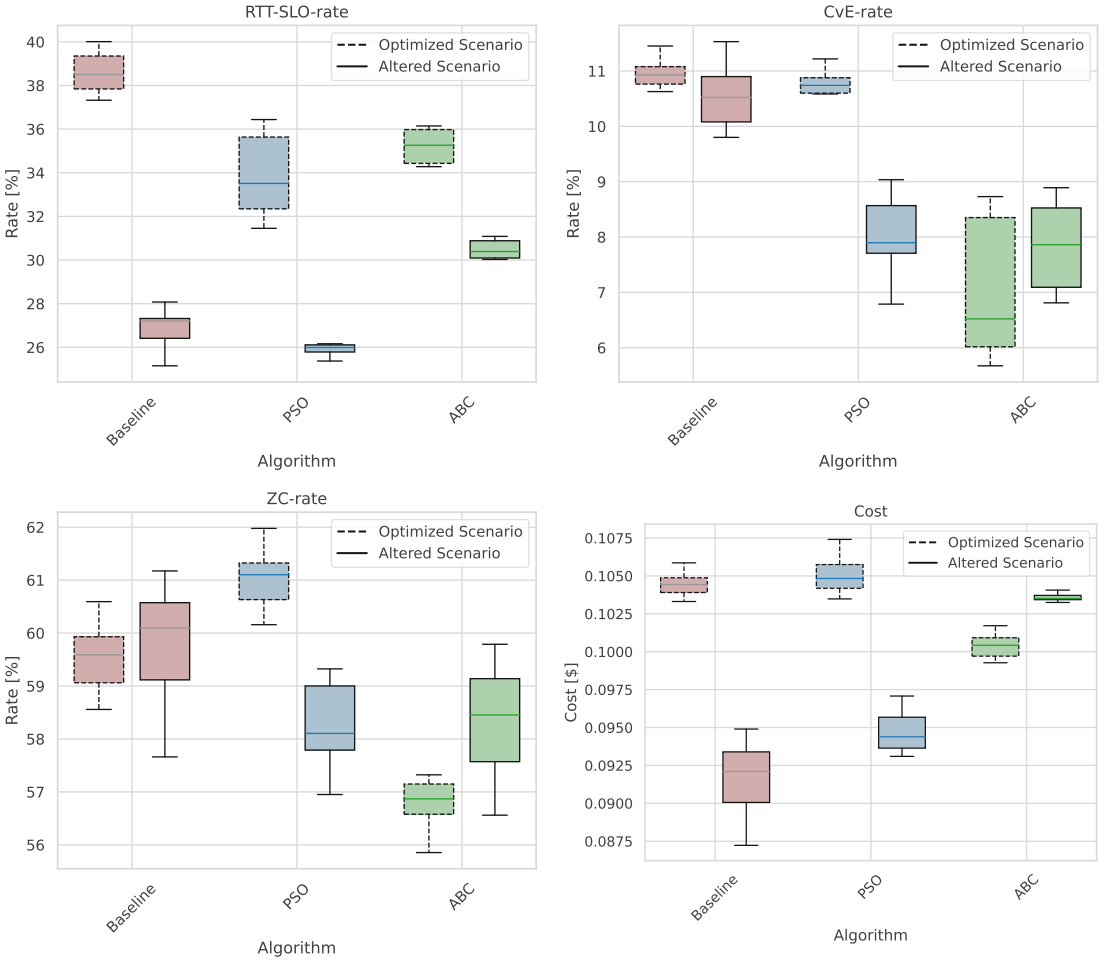

Figure 7.1: KPIs of setting with altered infrastructure compared to the optimized setting.

The increase in processing capabilities led to a noticeable decrease in RTT SLO violations across all configurations. The baseline configuration had the greatest improvement, showing a 29.3 % lower RTT-SLO-rate on the altered infrastructure. The PSO-discovered configuration also led to an improved rate in the altered setting. However, the relative improvement is slightly lower at 22.4 %. Despite that, the PSO-discovered configuration still beats the baseline on the altered infrastructure, with a median rate of 25.99 % compared to a median rate of 27.20 %. The ABC-discovered parameters also lead to an improved RTT-SLO-rate in the altered setting. However, absolute and relative improvements were the lowest observed at 0.05 and 13.8 % respectively. In the altered setting, the ABC parameter performed worse than the baseline with a median rate of 30.39 %.

CvE-rate only marginally improved by 3.7 % for the baseline parameters. This KPI greatly improved by 26.5 % for the PSO-discovered configuration. At a median rate of

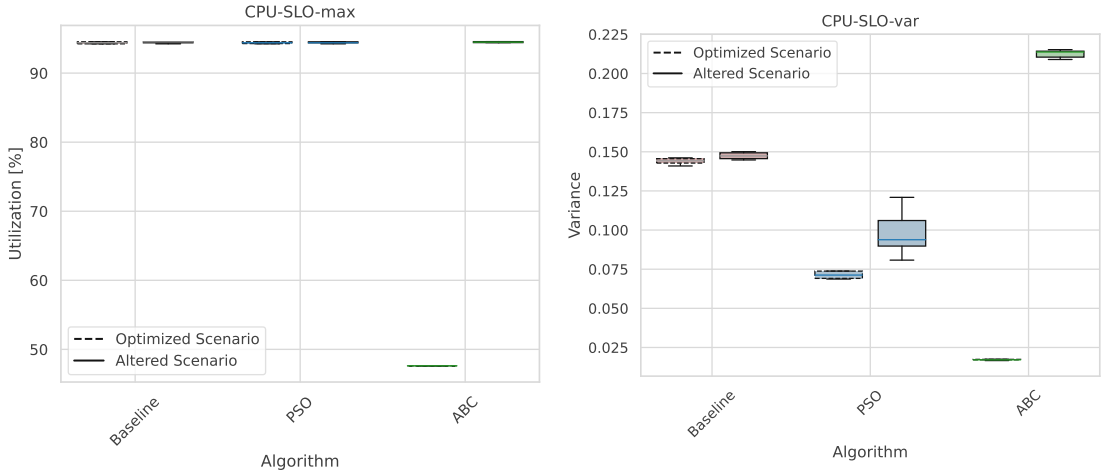

Figure 7.2: KPIs of setting with altered infrastructure compared to the optimized setting continued.

7.90 %, it remains ahead of the baseline, which shows a median rate of 10.52 %. The ABC-discovered configuration caused a 20.5 % increase in cloud processing, leading to performance comparable to the PSO parameters at a median rate of 7.86 %.

The number of zone crossings increased slightly for both the baseline and the ABC-discovered configuration, showing performance degradations of 0.85 % and 2.19 % respectively. This metric improved by 4.90 % when running the PSO-discovered parameter set on the altered infrastructure. So much so that it produced a median ZC-rate of 58.10 %, which is slightly lower than the median of ABC at 58.45 %, which beat it in the original setting.

Cost decreased for the baseline and PSO configurations by 11.80 % and 9.96 % respectively. The ABC-discovered configuration's cost increased by 3.08 %, making it higher than the other two with a median of 0.1035 $ compared to medians of 0.0920 $ for the baseline and 0.0944 $ for PSO.

CPU-SLO-max, which was maxed out on the original infrastructure for the baseline and PSO, was maxed out for all configurations on the altered infrastructure. The ABC-discovered parameters' ability to schedule resources in a way that decreased this metric to below 0.5 seems to have been completely lost when running them on the altered infrastructure.

Similarly, the ABC configuration lost a lot of quality regarding resource spread characterized by CPU-SLO-var with a degradation of 1150.25 %. On the original infrastructure, the ABC-discovered parameters caused the most evenly spread allocation of compute resources. However, on the altered infrastructure this configuration now shows the highest CPU-SLO-var with a median of 0.2137, even being beaten by the baseline parameter set with a median of 0.1473, which only slightly increased this KPI by 2.18 %. The

PSO-discovered parameters also slightly, but not dramatically increased this metric by 31.68 % leading to a median of 0.0938.

Overall, it seems like both optimized parameter sets were also a decent fit for the infrastructure with slightly more available resources regarding RTT-SLO-rate. However, only the PSO-discovered parameters also managed to show similar improvements to the baseline concerning CvE-rate, CZ-rate, and cost. Additionally, they showed decreases in performance regarding the CPU-based resource metrics comparable to the baseline while staying ahead of it. The ABC-discovered parameters seem to be sensitive to the infrastructure they are used on, particularly when it comes to resource-based KPIs. Hence, some form of overfitting may have occurred in their discovery process, while the PSO-discovered parameters appear to be quite robust with respect to slight changes in the infrastructure they are deployed on.

7.2 Differences in Workload Patterns

This experiment tests the performance of an edge-cloud platform using an autoscaler configuration optimized with an artificial workload when subjected to a more realistic request pattern.

7.2.1 Experimental Setup

The infrastructure used in this experiment is the one described in Section 6.1. To run simulations with a more realistic workload, data from the *Shanghai Telcom Dataset* [24] was used to derive eight request profiles for the different edge zones of the infrastructure. The profiles are therefore based on real user data that was made publicly available. The function called is the one from the original setting. The profiles are 3 minutes long and their RPS values have been selected, such that the total average load of all profiles combined once again roughly equals 80 % of the infrastructure's theoretical capacity without platform overhead. This mimics the load for which the parameters were tuned. Despite this approach, the total number of requests sent is higher. However, because the main aim here is to compare differences between configurations, this is accepted. The request profiles for each zone are visualized in Figures 7.3 and 7.4.

A simulation with the given setup was executed 10 times for each parameter set to compensate for the simulator's slight nondeterminism.

7.2.2 Results

Figures 7.5 and 7.6 show how the KPIs change for each parameter set when running simulations with the altered workload compared to the base setting, which was used to discover the configurations. Furthermore, Table 7.2 lists the absolute and relative differences of each KPI for each parameter set. For each KPI, the absolute and relative changes that represent either the best improvement of the metric or the smallest degradation, if no improvement was observed, are highlighted in bold. Analogously underlined values

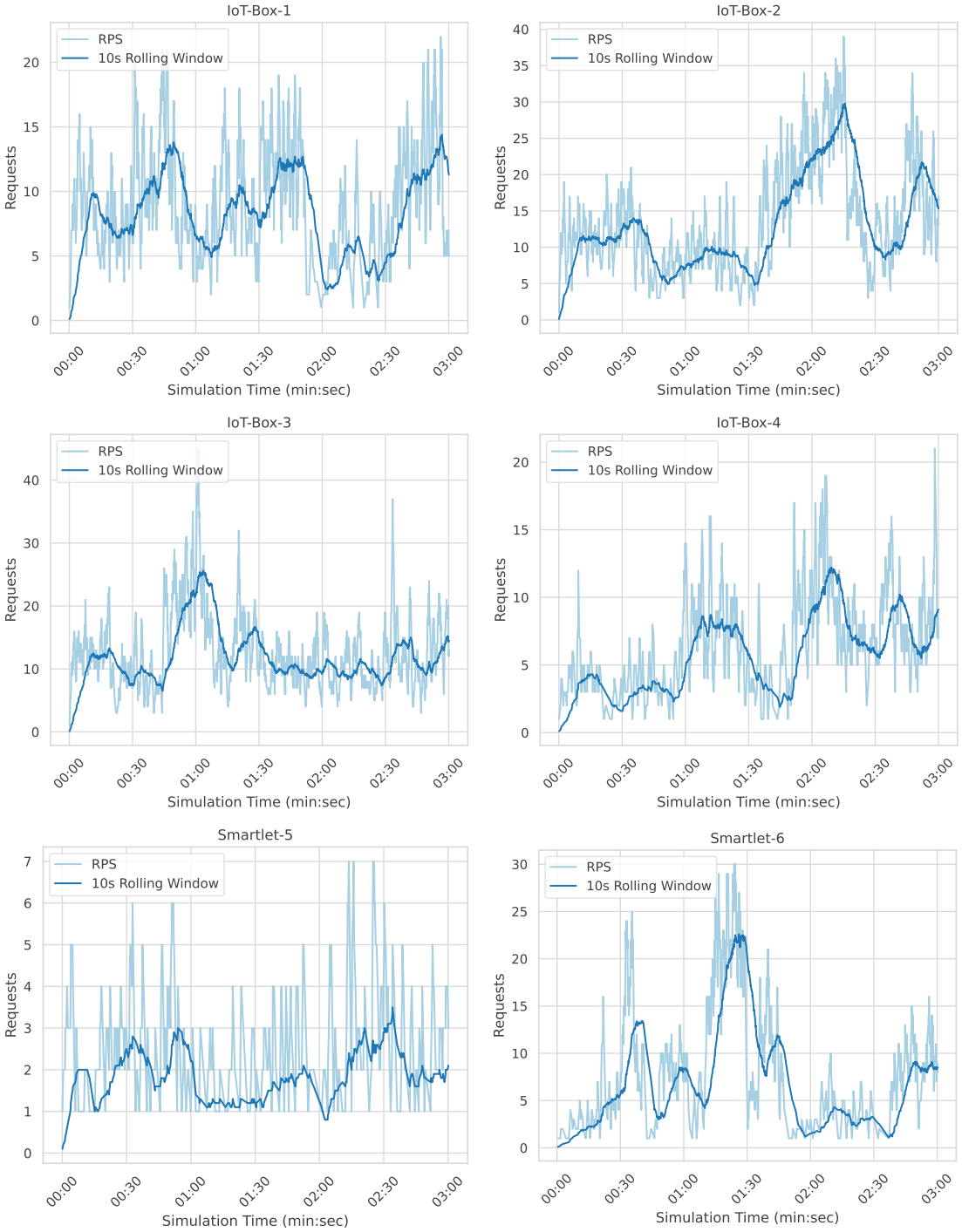

Figure 7.3: Request profiles used in robustness experiments regarding differences in workload patterns.

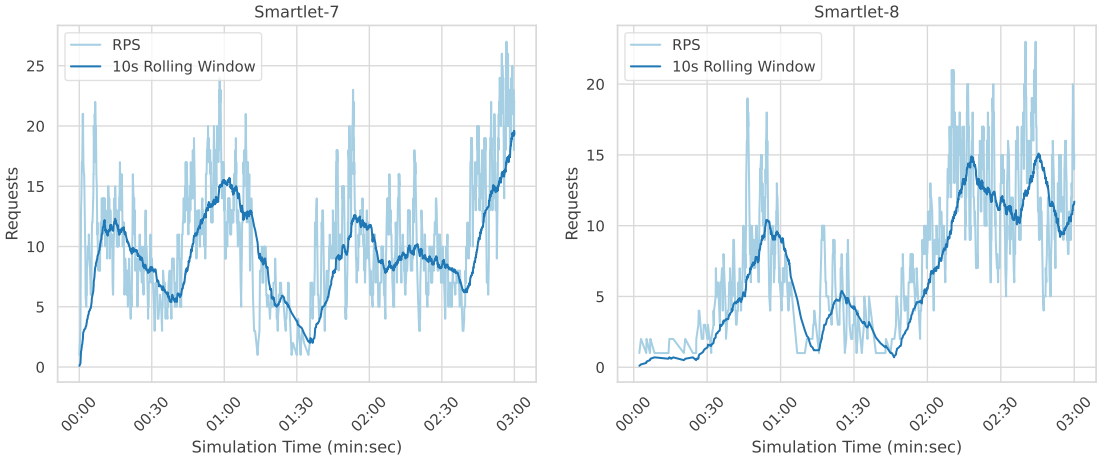

Figure 7.4: Request profiles used in robustness experiments regarding differences in workload patterns continued.

KPI	Baseline		PSO		ABC	
	absolute	relative	absolute	relative	absolute	relative
RTT-SLO-rate	**+0.241753**	**+62.7954 %**	+0.262713	+78.3957 %	+0.266706	+0.756349 %
CvE-rate	**+0.022674**	**+20.7452 %**	+0.029877	+27.8191 %	+0.056471	+86.5924 %
ZC-rate	−0.021930	−3.6802 %	**−0.026931**	**−4.4075 %**	+0.010088	+1.7738 %
Cost	**+0.782921**	**+749.7136 %**	+1.449496	+1382.6599 %	+0.871779	+868.0927 %
CPU-SLO-max	+0.001433	+0.1518 %	**+0.001396**	**+0.1479 %**	+0.467687	+98.2143 %
CPU-SLO-var	**−0.003541**	**−2.4555 %**	+0.015618	+21.9191 %	+0.194485	+1137.7372 %

Table 7.2: Changes in KPIs of setting with realistic workload compared to the optimized setting.

represent the biggest degradation or the smallest improvement if none occurred. As a basis for calculating the differences, the median value for each KPI across all simulation runs was used. The RAM-based metrics are omitted since they were always 0 across all parameter sets and scenarios.

The number of RTT SLO violations significantly increased for all configurations roughly in equal measure by 62.78 % for the baseline, 78.34 % for PSO and 75.56 % for ABC. Their relative order, when running with the synthetic workload, is maintained on the realistic workload. However, the differences between them shrink with the baseline configuration now showing a median rate of 62.67 %, the PSO-discovered one coming in at 59.78 % and ABC at 61.93 %.

The cloud processing rate also increased for all observed configurations. The baseline and PSO-discovered parameters led to a comparable degradation of 20.75 % and 27.82 % respectively. The most dramatic increase is observed for the ABC-discovered parameters at 86.56 %, which still perform the best despite this performance hit, showing a median rate of 12.17 %. The PSO-discovered configuration, which performed better than the baseline in the original setting, now performs the worst, showing a median rate of 13.73 %.

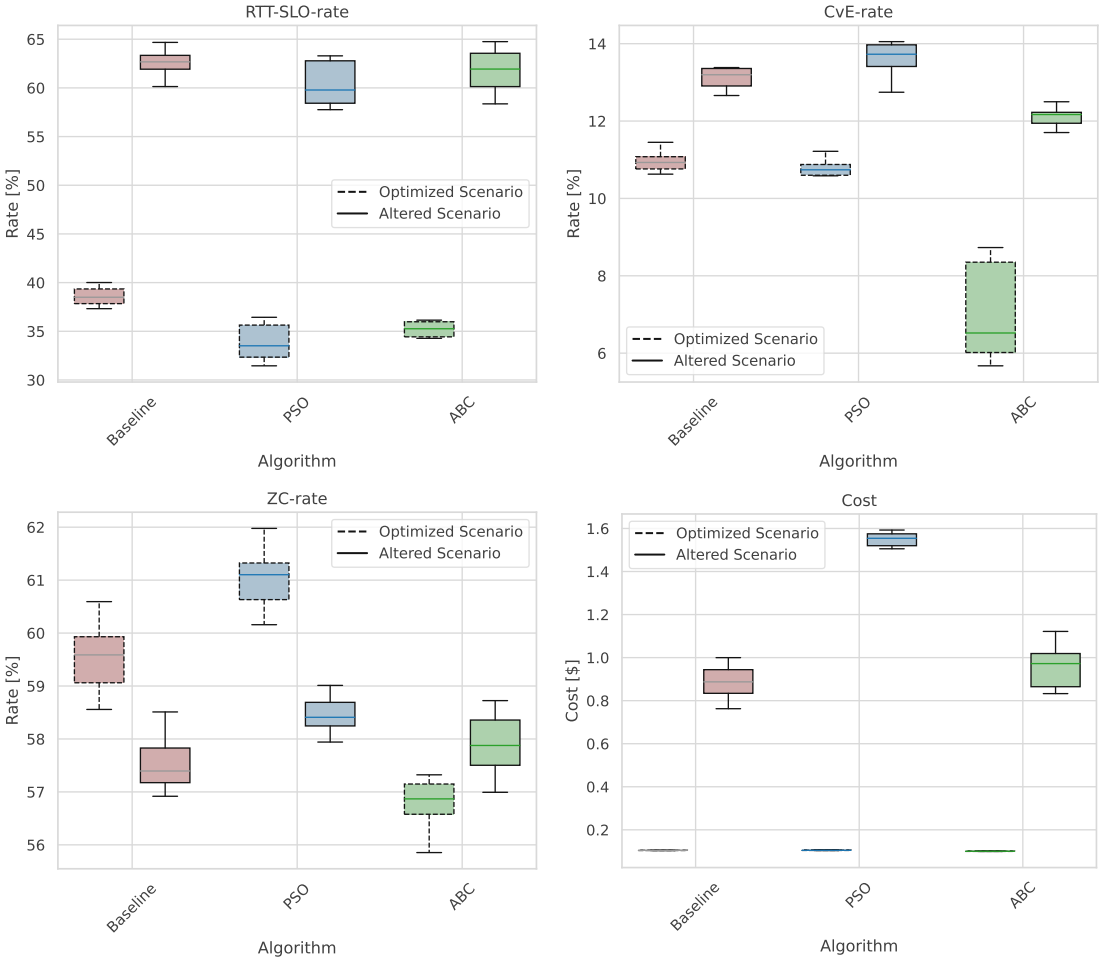

Figure 7.5: KPIs of setting with realistic workload compared to the optimized setting.

The number of zone crossings decreased for the baseline configuration and the one discovered by PSO, improving by 3.68 % and 4.41 %, respectively. The metric increased for the parameters discovered by ABC by 1.77 %. At a median rate of 57.88 %, they perform worse in the altered setting compared to the baseline, showing a rate of 57.39 %, but still better than PSO at a rate of 58.41 %. Overall, the differences in this metric between the three configurations greatly decreased when faced with the realistic workload.

Cost greatly increased across all three observed configurations. However, this is to be expected, as the total number of requests also increases when using the altered workload. The baseline configuration led to the smallest increase in cost of 749.71 %, followed by the ABC-discovered one at 1382.66 %. The PSO parameters caused the greatest cost hit, leading to a 13-fold increase, almost double that of the baseline parameters.

CPU-SLO-max shows the same pattern as previously observed for the experiments that altered the infrastructure. The ABC-discovered configuration seemingly lost its ability

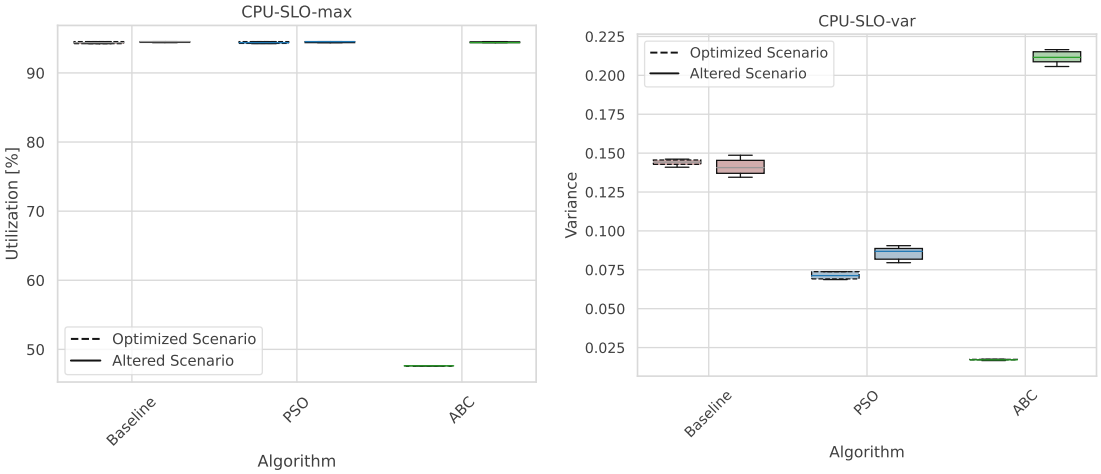

Figure 7.6: KPIs of setting with realistic workload compared to the optimized setting continued.

to greatly improve this metric and now sits on par with both the baseline and PSO, for which the metric was always at the maximum possible value across both settings.

CPU-SLO-var improved slightly for the baseline configuration by 2.46 %. The PSO-discovered parameters performed best in the altered setting, with a variance of 0.086872, but experienced a slight decrease in quality by 21.92 %. The ABC-discovered parameters, analogously to the experiment that altered the infrastructure, lose a lot of quality with respect to this metric, leading to a roughly 11-fold increase. They performed worst with respect to the realistic workload, now showing a variance of 0.211579. This once again hints towards this metric being the most volatile, if a parameter set has been specifically optimized for it.

Overall, the performance hits across most metrics experienced by the PSO-discovered configuration roughly match those of the baseline. An exception to this is cost, which scaled considerably worse. The ABC-discovered configuration caused large losses in performance across all observed metrics. Both configurations discovered by optimization performed worse than the baseline regarding multiple target metrics. This highlights the importance of accurate scenario modeling. Additionally, the results of the resource-based metrics once again showed that the ABC-discovered configuration, which had the overall best quality score during optimization, is quite volatile and may be the result of overfitting, at least regarding scheduled resources.

7.3 Differences in Load

This experiment tests the performance of an edge-cloud platform using an autoscaler configuration optimized for an expected load that differs from the actual load experienced.

Load	KPI	Baseline		PSO		ABC	
		absolute	relative	absolute	relative	absolute	relative
20 %	RTT-SLO-rate	**+0.003246**	**+0.8432 %**	+0.013876	+4.1408 %	+0.033175	+9.4082 %
	CvE-rate	**+0.014657**	**+13.4099 %**	+0.025864	+24.0819 %	+0.027388	+41.9962 %
	ZC-rate	**+0.011361**	**+1.9067 %**	+0.019278	+3.1550 %	+0.034071	+5.9911 %
	Cost	−0.063048	−60.3735 %	**−0.066268**	**−63.2126 %**	−0.059657	−59.4047 %
	CPU-SLO-max	+0.001433	+0.1518 %	**+0.001396**	**+0.1479 %**	+0.284210	+59.6841 %
	CPU-SLO-var	**−0.006649**	**−4.6102 %**	+0.029553	+41.4760 %	+0.099295	+580.8741 %
40 %	RTT-SLO-rate	+0.017713	+4.6009 %	+0.068077	+20.3146 %	**+0.006472**	**+1.8353 %**
	CvE-rate	**−0.005396**	**−4.9372 %**	+0.012069	+11.2376 %	+0.021147	+32.4268 %
	ZC-rate	+0.008462	+1.4201 %	**−0.005320**	**−0.8707 %**	−0.001188	−0.2089 %
	Cost	**−0.042857**	−41.0395 %	−0.042694	−40.7255 %	−0.042622	**−42.4415 %**
	CPU-SLO-max	**−0.005548**	**−0.5878 %**	+0.001396	+0.1479 %	+0.100733	+21.1538 %
	CPU-SLO-var	**+0.004245**	**+2.9431 %**	+0.016092	+22.5842 %	+0.019615	+114.7475 %
60 %	RTT-SLO-rate	−0.007149	−1.8571 %	+0.051513	+15.3719 %	**−0.012493**	**−3.5430 %**
	CvE-rate	**−0.005907**	**−5.4050 %**	−0.001833	−1.7068 %	+0.033077	+50.7207 %
	ZC-rate	+0.008341	+1.3999 %	**+0.003135**	**+0.5132 %**	+0.013789	+2.4247 %
	Cost	−0.028964	−27.7354 %	−0.027747	−26.4675 %	**−0.029815**	**−29.6891 %**
	CPU-SLO-max	−0.025474	−2.6989 %	−0.025510	−2.7027 %	**−0.091575**	**−19.2308 %**
	CPU-SLO-var	**+0.003131**	**+2.1711 %**	+0.015980	+22.4262 %	+0.004560	+26.6745 %

Table 7.3: Changes in KPIs of setting with lower load compared to the optimized setting.

7.3.1 Experimental Setup

The infrastructure used in this experiment is the one described in Section 6.1, and so is the deployed function. The workload profiles that are used mimic the ones presented in Section 6.1 in the sense that they are also sine wave-shaped, with the same frequencies per zone. However, they were generated with a different average RPS. The parameter sets subject to testing were optimized for a load representing 80 % of the infrastructure's theoretical capacity, disregarding potential platform overhead. Therefore, the workloads tested here were generated to represent 20 %, 40 %, 60 %, 100 % and 120 % of the theoretical capacity. Ideally, well-performing autoscaler configurations should act in a more resource-efficient way when faced with a low load. Loads higher than 80 % were included to simulate a stress test scenario.

A simulation with the given setup was executed 10 times for each parameter set and each target load, to compensate for the simulator's slight nondeterminism.

7.3.2 Results

Figures 7.7 and 7.8 show how the KPIs change for each parameter set, when running simulations with the altered loads. Load increases for each configuration from left to right in increments of 20 %. The data for 80 % load, visualized in the box with solid lines, represents the scenario for which the configuration was optimized. Furthermore, Table 7.3 lists the absolute and relative differences of each KPI for each parameter set run on loads lower than the one optimized for. Table 7.4 lists the same data for loads that were higher than the one optimized for. For each load and each KPI, the absolute and relative changes that represent either the best improvement of the metric or the smallest

Load	KPI	Baseline		PSO		ABC	
		absolute	relative	absolute	relative	absolute	relative
100 %	RTT-SLO-rate	−0.017161	−4.4577 %	+0.049697	+14.8299 %	**−0.044401**	**−12.5916 %**
	CvE-rate	+0.009117	+8.3417 %	**+0.005895**	**+5.4891 %**	+0.046671	+71.5655 %
	ZC-rate	+0.034195	+5.7386 %	**+0.016166**	**+2.6458 %**	+0.047002	+8.2650 %
	Cost	+0.009557	+9.1517 %	+0.013612	+12.9844 %	**+0.005368**	**+5.3452 %**
	CPU-SLO-max	+0.000037	+0.0039 %	+0.000285	+0.0302 %	**−0.091575**	**−19.2308 %**
	CPU-SLO-var	+0.002912	+2.0189 %	**−0.006814**	**−9.5637 %**	+0.009058	+52.9908 %
120 %	RTT-SLO-rate	**+0.002117**	**+0.5499 %**	+0.050364	+15.0291 %	+0.046436	+13.1689 %
	CvE-rate	+0.006729	+6.1570 %	**−0.004682**	**−4.3593 %**	+0.045553	+69.8513 %
	ZC-rate	+0.022544	+3.7833 %	**−0.007195**	**−1.1776 %**	+0.037301	+6.5592 %
	Cost	+0.052132	+49.9205 %	**+0.046913**	**+44.7502 %**	+0.045688	+45.4948 %
	CPU-SLO-max	+0.000037	+0.0039 %	**−0.001469**	**−0.1557 %**	+0.100733	+21.1538 %
	CPU-SLO-var	−0.000526	−0.3647 %	**−0.005656**	**−7.9383 %**	+0.010888	+63.6937 %

Table 7.4: Changes in KPIs of setting with higher load compared to the optimized setting.

degradation, if no improvement was observed, are highlighted in bold. Analogously underlined values represent the biggest degradation or the smallest improvement if none occurred. As a basis for calculating the differences, the median value for each KPI was used. The RAM-based metrics are omitted since they were always 0 across all parameter sets and scenarios.

Surprisingly, the baseline's RTT SLO violations seem to stay very consistent at higher loads, indicating that this configuration is suited for more stressful situations. However, this is only a rough trend and there are slight fluctuations, causing the configuration to perform best for 100 % load, with a median rate of 36.78 %, and worst for 40 % load, with a rate of 40.26 %. The PSO-discovered parameters cause a significantly larger amount of fluctuation without a clear pattern. The parameters perform best under 80 % load, which they were optimized for, but fail to beat the baseline regarding 40 %, 60 % and 100 % load, showing rates of 40.31 %, 38.66 %, and 38.48 % respectively. The ABC-discovered configuration appears to generally increase in quality with higher load, with the exception of the 120 % stress test scenario, where the RTT-SLO-rate suddenly spikes to a median of 39.91 %. In general, the ABC-discovered parameters performed better than the baseline, except for the 120 % load scenario. The overall range of values observed was noticeably smaller compared to the experiment altering workload patterns.

Cloud processing of the baseline parameters follows a V-shape, with the best median rate of 10.34 % achieved at 60 % load and the worst rates observable at very low and very high loads. The CvE-rate of the PSO-discovered configuration appears to be particularly poor for low loads, landing behind the baseline for both 20 % and 40 % at median rates of 13.33 % and 11.95 %, respectively. For all other loads, it remained relatively consistent and beats the baseline for 100 % and 120 % with rates of 11.33 % and 10.27 %. The ABC-discovered configuration had consistently low CvE-rates for loads below and at the load that it was optimized for. Values diverge upwards for loads higher than 80 %. However, the CvE-rate consistently stays below that of the baseline and PSO-discovered parameters, except for a spike at 120 % load manifesting in a median rate of 11.08 %. The observed value ranges are very close to those of the workload-altering experiment.

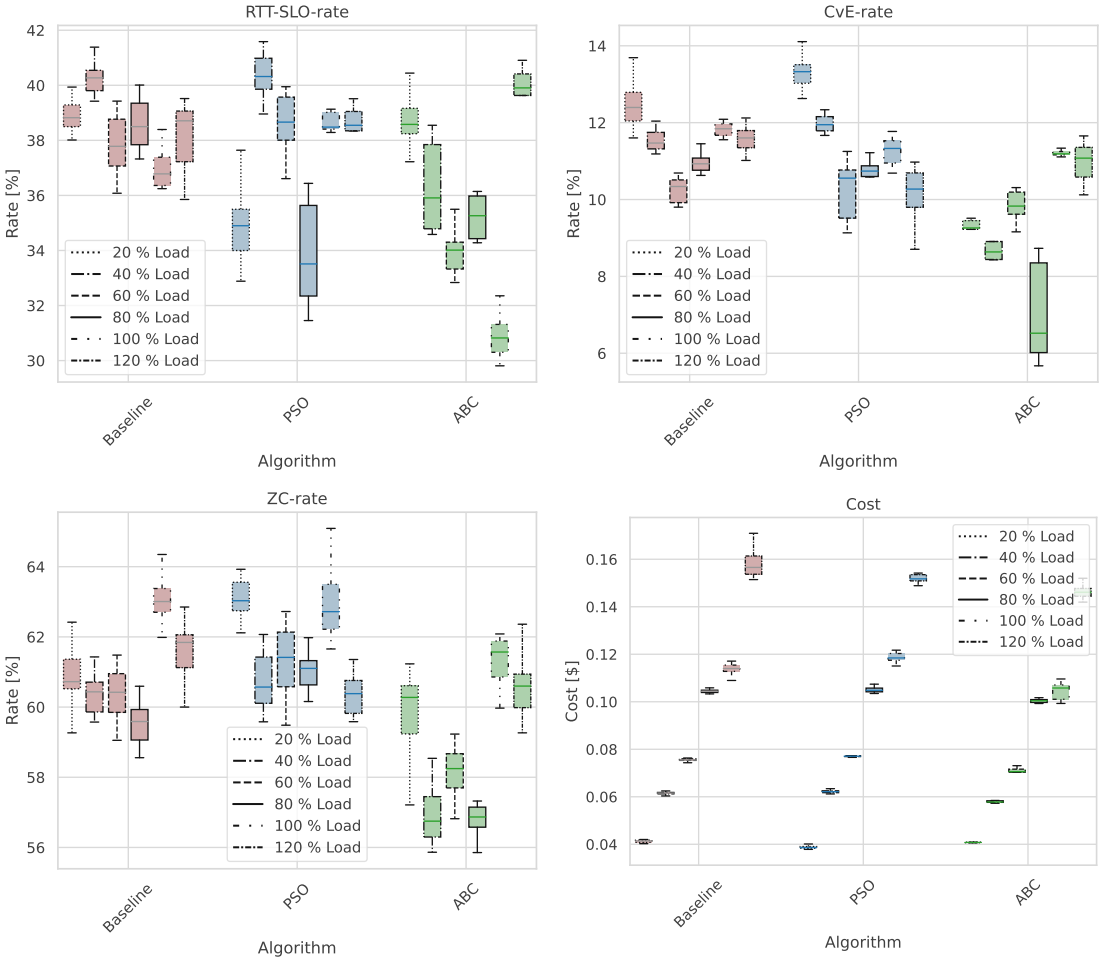

Figure 7.7: KPIs of setting with altered loads compared to the optimized setting.

The ZC-rate of the baseline remained very consistent for loads between 20 % and 80 %. For the stress test scenarios of 100 % and 120 % it slightly increased by 5.74 % and 3.78 %, respectively, leading to median rates of 63.01 % and 61.84 %. The PSO-discovered parameter set remained slightly above the baseline for 40 % and 60 % load at median rates of 60.57 % and 61.42 %. This is to be expected, as this configuration, in general, seems to neglect this KPI in favor of others. For high loads of 100 % and 120 %, the configuration performs better than the baseline at rates of 62.72 % and 60.38 %. For a very low load of 20 %, it performs worse at a rate of 63.03 %. The ABC-discovered configuration shows a pattern very similar to that of the PSO counterpart. However, the values are generally much lower and the fluctuations between loads are more pronounced. Across all loads, this configuration consistently performed better than the baseline and is only beaten by PSO at 120 %, where it achieved a median rate of 60.56 %.

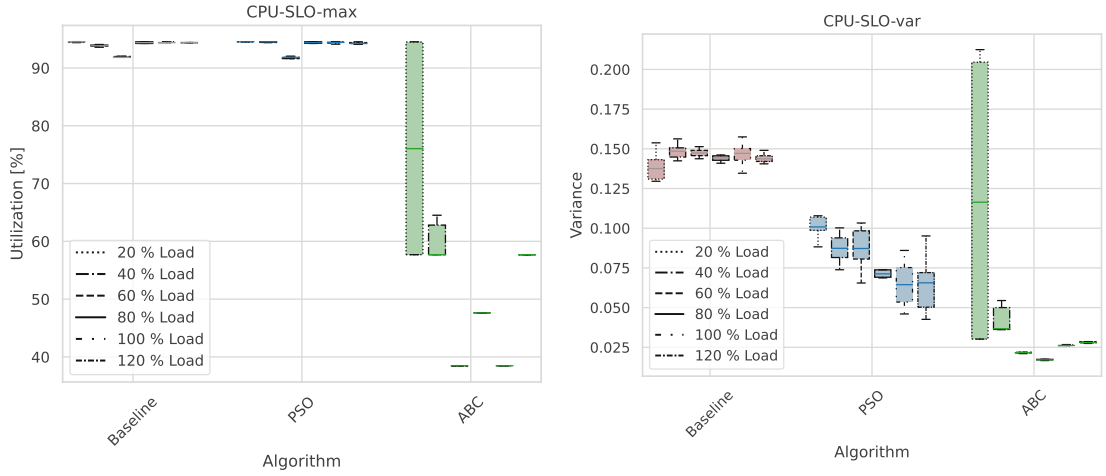

Figure 7.8: KPIs of setting with altered loads compared to the optimized setting continued.

Cost developed very homogeneously across all three observed configurations. As expected, cost scaled relative to the total load. Interestingly, the smallest observable jump in cost across all three parameter sets happened between 80 % and 100 % load. Notably, the baseline scaled the worst, when faced with a very high load of 120 %, leading to a 49.92 % performance degradation, while the PSO-discovered parameters experienced the smallest performance drop at 44.75 %. The ABC-discovered configuration kept costs at that load the lowest at 0.1461 $.

Concerning CPU-SLO-max, the baseline parameters caused this KPI to remain maxed out for all, but 40 % and 60 % loads. However, for those, the metric still remained above 90 %. The PSO-discovered configuration only managed to lower the rate for 60 % load. The ABC-discovered configuration remained consistently good across all loads. A slight exception is presented at 20 %, where there appear to be large fluctuations between simulation runs with the same parameters. In the cases of 60 % and 100 % load, this configuration consistently lowered the metric below 0.4, a value not observed among any other conducted experiments.

CPU-SLO-var remained very consistent but high for the baseline. The KPI improved for the PSO-discovered parameters for higher loads. Furthermore, the values remained well below those of the baseline, only dipping above 0.1 at 20 % load. The results regarding ABC paint a similar picture as the other CPU-based metric, where the lowest load caused a large amount of fluctuation of that metric, while all other loads kept it consistently below 0.05.

Overall, there is a noticeable difference between the KPIs and the way they develop regarding loads and different parameter sets. Some KPIs, such as cost, scaled as expected, while other KPIs, such as CvE-rate, showed a more chaotic pattern. Based on this fact, one can argue that it might be beneficial to optimize parameters for multiple loads and choose those configurations that on average perform well across all. Alternatively, one

could optimize configurations for different loads and exchange them at runtime. The ABC-discovered configuration performed far more consistently well for this set of experiments compared to those that altered infrastructure or workload patterns. However, it did struggle to perform consistently at very high loads and showed the overall worst relative degradations with respect to the KPIs. The PSO parameters only outperformed the baseline concerning some metrics, such as RTT-SLO-rate and CPU-SLO-var, but showed comparable or worse behavior when looking at others, like CZ-rate and CPU-SLO-max. However, at high loads, they experienced the lowest performance degradation.

7.4 Key Observations

Looking at the three conducted experiments, the following key observations can be made about the results.

- Optimized configurations appear quite robust to minor infrastructure changes.
- Optimized configurations were not very robust when faced with a workload that had very different patterns and load distributions compared to the optimized scenario. This emphasizes the need for accurate request modeling, at least in terms of distribution across zones.
- Optimized configurations showed moderate robustness to load changes. However, differing KPI trends under varying loads suggest incorporating load changes into either the optimization process or runtime adjustments.
- The PSO configuration, representing an average-quality solution, fluctuated slightly more than the baseline across different setup permutations. However, these fluctuations are insufficient to deem the parameters unfit due to overfitting. The parameters were able to make use of additional compute resources, but not lower loads.
- The ABC configuration, an outlier with a particularly good quality score, showed larger KPI fluctuations across different setup permutations. It was especially sensitive to changes in infrastructure and very high loads. However, for most KPIs, this configuration still outperformed PSO and the baseline in most cases.
- The most fluctuating metrics were resource utilization-based. This suggests that they might not be ideal target metrics, potentially leading to unwanted overfitting.

CHAPTER 8

Conclusion

This chapter concludes the thesis with a brief summary of the work, a discussion of the results in relation to the initial research questions, and a look at potential future research directions.

8.1 Summary

To address various limitations of the cloud computing paradigm, edge computing has been introduced as a promising computing model, which allows processing to happen closer to the end user. A serverless service model is a natural fit for this setting. The need to configure static autoscaling parameters was introduced as one of the various obstacles holding back novel edge computing solutions from reaching their full potential. The thesis's goal of exploring ways to automatically optimize these parameters before the deployment of an edge-cloud solution using a simulator was motivated alongside the choice to focus efforts on metaheuristic optimization.

To allow an optimization process to take place, the need to gauge platform runtime performance was rationalized. A literature review was conducted to discover the most commonly utilized metrics for measuring platform orchestration quality. This list was distilled down to a set of key metrics of interest. Relationships between these metrics were explored to identify a set of performance indicators with minimal information redundancy. A minimum-weight subgraph approach was used to select eight distinct KPIs, which were used in the remainder of the work. Additionally, an aggregate function, distilling them down into a single quality score was presented to be used for single-objective optimization. While the rationale behind the choices was provided, it was emphasized that no single KPI set or quality function is perfect, and the definition of quality should remain flexible in this context.

The goal of optimizing the parameters of a selected autoscaling solution was formalized as a single-objective and a multi-objective optimization problem. To solve it, six metaheuristic algorithms were implemented and adapted to be applicable in the given setting. The algorithms chosen were Particle Swarm Optimization, the Genetic Algorithm, Artificial Bee Colony, Differential Evolution, and Cuckoo Search Optimization. Additionally, NSGA-II was explored as a representative of multi-objective optimization algorithms. A list of all hyperparameters of each algorithm was also provided.

An experiment was conducted in which the autoscaler configuration of a small-scale smart city deployment was optimized using each algorithm. The results showed PSO and ABC as the top performers. NSGA-II underperformed, confirming suspicions about the limited applicability of multi-objective optimization algorithms for the given setting.

The viability of optimizing edge-cloud autoscaling configurations was explored via three experiments on two optimized parameter sets and a baseline. Each one altered a certain aspect of the setting. While optimized configurations performed acceptably and often outperformed the baseline, certain shortcomings were identified and discussed.

8.2 Discussion

The results of the research activities are discussed below in the context of the research questions posed in Chapter 1.

8.2.1 Research Question 1

How can the quality of an edge-cloud deployment best be estimated based on available runtime metrics to guide an orchestration parameter optimization process?

As the conducted literature review shows, there exists a wide variety of metrics used among published work to gauge the quality of edge-cloud orchestration mechanisms at runtime. Different KPIs often relate to competing aspects of the system. There is no definitive group of metrics that can be considered a superior subset.

Experimental exploration of the relationships between the collected metrics showed that there exists clear information overlap between some of them. A systematic approach was introduced as a possible way to arrive at a representative set with minimal information redundancy. It involves calculating correlation coefficients between metrics from sample deployment scenarios and then selecting a subgraph with minimum edge weight, where the weight is the correlation between two given metrics. The metrics selected by this approach were:

- the round-trip-time-based SLO violation rate (RTT-SLO-rate),
- the ratio of requests that were offloaded to the cloud (CvE-rate),
- the ratio of requests processed outside of the originating edge zone (ZC-rate),

- the estimated operational cost based on AWS Lambda@Edge pricing (cost),
- the highest CPU-utilization-based SLO violation rate among all nodes (CPU-SLO-max) and its RAM-based counterpart (RAM-SLO-max), and
- the variance of the CPU-utilization-based SLO violation rate between nodes (CPU-SLO-var) and its RAM-based counterpart (RAM-SLO-var).

It is important to emphasize that this set of metrics is simply the result of the presented approach and no claims are made that it is in any way better than any other. The experiments presented in Chapter 7 indicate that the choice of resource metrics may not have been ideal, as these KPIs were the most susceptible to changes in the setting. Furthermore, these metrics performed poorly with the simulator's slight nondeterminism, sometimes showing large differences between runs with identical parameters. To enable the use of single-objective optimization algorithms, an aggregate quality function was introduced in the form of a weighted sum, which tried to strike a balance between competing aspects of the system and stakeholder interests. Although this approach has theoretical drawbacks, it proved to be quite practical for the given setting, as almost all single-objective approaches outperformed the explored multi-objective metaheuristic NSGA-II during the experiments described in Chapter 6. While aggregating metrics into a single score has potential drawbacks, the experiments demonstrated its superiority to multi-objective approaches in this context due to the large number of competing goals.

Ultimately, there is no single correct way to objectively measure edge orchestration quality, and this aspect should always remain open to changes based on concrete stakeholder interests. Hence, all optimization schemes that were explored kept this aspect flexible. This is also something that is highly recommended for any potential future work based on the presented contributions.

8.2.2 Research Question 2

Among promising optimization techniques, which performs best when used to optimize bootstrapping parameters of edge-cloud autoscaling solutions?

The optimizers' performance was evaluated by having each algorithm tune the autoscaler parameters of a benchmark scenario, modeled after a small smart city deployment using the presented quality definition. The utilized workload was synthetically generated to mimic sine wave patterns as this shape was deemed reasonably realistic. Key performance metrics included the final configuration's quality score, the number of simulator runs required, and the number of iterations taken. A baseline of an arbitrary but reasonable configuration was chosen to compare against.

The results were presented in Chapter 6. No single algorithm dominated all performance metrics. While most discovered configurations across all algorithms had associated quality scores that were similar to each other, DE and ABC discovered two outliers with significantly better values. Observing the associated orchestration metrics hints that this

possibly happened due to suboptimal resource metrics, not necessarily poor optimizer performance.

ABC and PSO performed best with respect to the quality of their discovered configurations, improving the baseline on average by 8.6 % and 7.1 % respectively. PSO was more efficient, taking on average 525 simulation runs to converge compared to ABC's 1205. Furthermore, the results of PSO were more consistent, showing a standard deviation of 0.002295 among quality scores, compared to a standard deviation of 0.016925 for ABC.

The GA underperformed, only improving the quality score by 5.9 % on average. DE, the other evolutionary algorithm evaluated, performed similarly to PSO in terms of the average improvement in quality score over the baseline, which comes in at 7.0 %. However, the results were less consistent, showing a standard deviation of 0.010337. One of the aforementioned outliers was discovered by DE. Looking at the shape of the curve obtained from plotting quality to executed simulation runs, it seems likely that this discovery represents more of a lucky hit than a consistently achievable result. CSO proved to be entirely unfit for the given scenario, only improving the quality score by an average of 2.0 % compared to the baseline. This is most likely because CSO cannot handle a highly constrained search space well.

NSGA-II, as the only multi-objective optimization scheme evaluated, also underperformed. It did not significantly improve any of the eight observed orchestration KPIs compared to the baseline and the other algorithms. A possible reason for this is the high dimensionality of the search space combined with the high number of objectives, which makes it hard for NSGA-II to find a representative Pareto front.

A clear recommendation can be made in favor of the PSO if the goal is to arrive at a decent set of autoscaling parameters with a reasonable computational effort. If algorithmic runtime and computational cost are not of concern, ABC is most likely the better choice. Furthermore, because the resource-based metrics in the quality function were found to have drawbacks, optimizer performance could possibly improve with better alternatives. However, this is unlikely to change the overall recommendations in favor of PSO and ABC.

Finally, the main challenge of performing optimization in the given setting proved to be the expensive quality function. Simulator performance improvements, or the use of an alternative simulator, could enable the exploration of approaches that are currently infeasible.

8.2.3 Research Question 3

How robust are the parameters resulting from such a scheme when faced with fluctuations in the infrastructure, request patterns, and load at runtime?

Experiments presented in Chapter 7 showed that optimized autoscaler configurations often performed decently with respect to changes in the underlying scenario. However, this was not always the case and certain changes had more impact than others. The

experiments compared a PSO-discovered configuration and the ABC-discovered outlier with the same baseline configuration that was already used during optimizer evaluation.

Changes in the underlying infrastructure were compensated very well by the PSO-discovered parameters, which showed changes in the orchestration KPIs similar to or better than those of the baseline. In the given test scenario, ABC could not make efficient use of the extra resources that were made available on the altered infrastructure, only slightly improving SLO violations and losing performance with respect to all other KPIs. In particular, the resource-based KPIs suffered with a 98.4 % higher CPU-SLO-rate and an 1150.3 % higher CPU-SLO-var on the altered infrastructure. This may indicate that a high-quality score does not necessarily reflect better portability to different infrastructures, and overfitting in this regard may occur during optimization. Alternatively, this could also have been caused by the choice resource KPIs, which were particularly good for the ABC-discovered parameters.

The biggest impact among changed aspects was caused by the use of request patterns derived from real user data as opposed to synthetic ones. The PSO- and ABC-discovered parameters experienced higher negative impacts on most KPIs compared to the baseline. Particularly outstanding examples include an 86.6 % worse CvE-rate for the ABC-discovered configuration and a 1382.7 % increase in cost for the PSO-discovered one. This emphasizes the importance of accurate modeling of the expected user behavior. The impact seems to come primarily from the shape and load distribution of the request patterns, as the experiments that altered the overall system load did not cause nearly as much degradation among quality metrics.

A notable observation made during experiments with altered loads is that the development of KPIs regarding load fluctuations seems to be very hard to predict. There was also a notable difference in behavior between the configurations discovered by PSO and ABC and the chosen baseline. For example, the RTT-SLO-rate of PSO-discovered parameters got worse at every load that was not the one for which the parameters were optimized, while the ABC-discovered parameters achieved their best result concerning this KPI at 60 % and 100 % load. This motivates the exploration of approaches that perform optimization at different loads or swap configurations at runtime.

8.3 Future Work

Based on the obtained results, several promising future research avenues have been identified and are outlined in this section.

First, since the chosen resource-based quality metrics were shown to have downsides, further research on representative metrics for resource utilization and distribution is needed. Additionally, comparing configurations optimized with a different formulation of resource utilization quality would provide valuable insight into the applicability and portability of the presented approach.

Furthermore, since PSO, alongside ABC, performed best among the evaluated metaheuristics, it would be interesting to also explore how well extensions of the canonical PSO perform. Examples include Accelerated PSO and Adaptive PSO. PSO-EA would also be an interesting candidate algorithm as an example of a hybrid approach. Exploring the impact of PSO's algorithm-specific constraint handling capabilities is another promising avenue for future research.

This thesis only explored six metaheuristics which were primarily selected with diversity among them in mind. Potential future work could use the evaluation approach presented here to explore other potential optimization algorithms. These could also include different multi-objective methods, which were intentionally not the focus of the presented research activities.

Optimizer evaluation was conducted using a single scenario which was created to represent a small-scale smart city edge-cloud deployment. Future research activities could include the creation of a benchmarking suite that also includes other scenarios, for example, a deployment inspired by an industry 4.0 use case.

One of the main challenges of performing optimization in the presented scenario is the costly quality function. Improving the utilized simulator could open up more possibilities with regard to the available algorithms.

This thesis focused on one autoscaling solution benefiting from parameter tuning. Applying this approach to other autoscaling solutions with different configurations is also clearly of interest. Furthermore, the described approaches could be used to optimize other edge-cloud orchestration aspects, such as load balancing and scheduling.

Since the experiments presented in Chapter 7 showed that different loads on the system have a hard-to-predict impact on the orchestration KPIs, it would be worth exploring an alternative optimization approach, where each candidate configuration is evaluated based on multiple simulations with altering load. It would be interesting to explore whether such an approach would reduce the impact of running optimized parameters at different loads.

In a similar sense, one could also imagine a system that uses multiple sets of parameters, each optimized with a different load, that are then exchanged at runtime depending on the actual experienced load on the system. The evaluation of the viability of such a scheme would represent an interesting extension of the research activities presented here.

Lastly, since exchanging the synthetic workload with one based on real user data created the largest differences in the performance of the optimized configurations, future work could focus on possibilities to bridge this gap and mitigate this problem. The results presented in this thesis could then be used to compare a novel approach against.

Overview of Generative AI Tools Used

- **Google Gemini**[1] was used to answer questions related to grammar, phrasing, spelling, and general best practices for scientific writing. No prompt outputs were directly incorporated in the thesis.
- **Writeful**[2] was used to check spelling, grammar, and writing style. The tool is directly incorporated into Overleaf[3], the IDE used to write the Latex document.

[1] https://gemini.google.com/app
[2] https://www.writefull.com
[3] https://www.overleaf.com

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Ilyos Abdullaev et al. "Task Offloading and Resource Allocation in IoT Based Mobile Edge Computing Using Deep Learning". In: *Computers, Materials & Continua* 76.2 (2023), pp. 1463–1477. ISSN: 1546-2226. DOI: 10.32604/cmc.2023.038417. URL: https://www.techscience.com/cmc/v76n2/53982 (visited on 02/15/2025).

[2] Mainak Adhikari, Satish Narayana Srirama, and Tarachand Amgoth. "Application Offloading Strategy for Hierarchical Fog Environment Through Swarm Optimization". In: *IEEE Internet of Things Journal* 7.5 (May 2020), pp. 4317–4328. ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2019.2958400. URL: https://ieeexplore.ieee.org/document/8931777/ (visited on 02/14/2025).

[3] Ishtiaq Ahammad. "Fog Computing Complete Review: Concepts, Trends, Architectures, Technologies, Simulators, Security Issues, Applications, and Open Research Fields". In: *SN Computer Science* 4.6 (Oct. 4, 2023), p. 761. ISSN: 2661-8907. DOI: 10.1007/s42979-023-02235-9. URL: https://doi.org/10.1007/s42979-023-02235-9 (visited on 02/15/2025).

[4] Samson Busuyi Akintoye and Antoine Bagula. "Improving Quality-of-Service in Cloud/Fog Computing through Efficient Resource Allocation". In: *Sensors* 19.6 (Jan. 2019). Number: 6 Publisher: Multidisciplinary Digital Publishing Institute, p. 1267. ISSN: 1424-8220. DOI: 10.3390/s19061267. URL: https://www.mdpi.com/1424-8220/19/6/1267 (visited on 02/15/2025).

[5] Mohammad S. Aslanpour, Sukhpal Singh Gill, and Adel N. Toosi. "Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research". In: *Internet of Things* 12 (Dec. 1, 2020), p. 100273. ISSN: 2542-6605. DOI: 10.1016/j.iot.2020.100273. URL: https://www.sciencedirect.com/science/article/pii/S2542660520301062 (visited on 02/14/2024).

[6] Mohammad S. Aslanpour et al. "Serverless Edge Computing: Vision and Challenges". In: *Proceedings of the 2021 Australasian Computer Science Week Multiconference*. ACSW '21. New York, NY, USA: Association for Computing Machinery, Feb. 1, 2021, pp. 1–10. ISBN: 978-1-4503-8956-3. DOI: 10.1145/3437378.3444367. URL: https://dl.acm.org/doi/10.1145/3437378.3444367 (visited on 02/15/2025).

[7] Munish Bhatia, Sandeep K. Sood, and Simranpreet Kaur. "Quantum-based predictive fog scheduler for IoT applications". In: *Computers in Industry* 111 (Oct. 1, 2019), pp. 51–67. ISSN: 0166-3615. DOI: 10.1016/j.compind.2019.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0166361519301402X (visited on 02/15/2025).

[8] Bernd Bischl et al. "Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges". In: *WIREs Data Mining and Knowledge Discovery* 13.2 (2023). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1484, e1484. ISSN: 1942-4795. DOI: 10.1002/widm.1484. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1484 (visited on 02/15/2024).

[9] Sebastian Böhm and Guido Wirtz. "A Quantitative Evaluation Approach for Edge Orchestration Strategies". In: *Service-Oriented Computing*. Ed. by Schahram Dustdar. Cham: Springer International Publishing, 2020, pp. 127–147. ISBN: 978-3-030-64846-6. DOI: 10.1007/978-3-030-64846-6_8. (Visited on 02/15/0205).

[10] Xindi Cai et al. "Time series prediction with recurrent neural networks trained by a hybrid PSO–EA algorithm". In: *Neurocomputing*. Selected papers from the 3rd International Conference on Development and Learning (ICDL 2004) 70.13 (Aug. 1, 2007), pp. 2342–2353. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2005.12.138. URL: https://www.sciencedirect.com/science/article/pii/S0925231207000380 (visited on 02/15/2025).

[11] Claudia Canali and Riccardo Lancellotti. "GASP: Genetic Algorithms for Service Placement in Fog Computing Systems". In: *Algorithms* 12.10 (Oct. 2019). Number: 10 Publisher: Multidisciplinary Digital Publishing Institute, p. 201. ISSN: 1999-4893. DOI: 10.3390/a12100201. URL: https://www.mdpi.com/1999-4893/12/10/201 (visited on 02/15/2025).

[12] Paul Castro et al. "The rise of serverless computing". In: *Commun. ACM* 62.12 (Nov. 21, 2019), pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454. URL: https://dl.acm.org/doi/10.1145/3368454 (visited on 02/15/2025).

[13] Xianfu Chen et al. "Performance Optimization in Mobile-Edge Computing via Deep Reinforcement Learning". In: *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*. ISSN: 2577-2465. Aug. 2018, pp. 1–6. DOI: 10.1109/VTCFall.2018.8690980. URL: https://ieeexplore.ieee.org/abstract/document/8690980 (visited on 02/15/2025).

[14] Houda Chouat, Imed Abbassi, and Mohamed Graiet. "A genetic-based requirements-aware approach for reliable IoT applications in the Fog". In: *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. ISSN: 2641-8169. Bayonne, France, Oct. 2021, pp. 39–44. DOI: 10.1109/WETICE53228.2021.00019. URL: https://ieeexplore.ieee.org/document/9680484 (visited on 02/15/2025).

[15] Kenneth Alan De Jong. "An analysis of the behavior of a class of genetic adaptive systems". PhD thesis. Michigan: University of Michigan, 1975.

[16] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002), pp. 182–197. ISSN: 1941-0026. DOI: 10.1109/4235.996017. URL: https://ieeexplore.ieee.org/document/996017 (visited on 02/15/2024).

[17] Flexera. *Flexera State of the Cloud Report 2024*. Germany: Flexera Software LLC, 2024.

[18] Ahmed Fawzy Gad. “PyGAD: an intuitive genetic algorithm Python library”. In: *Multimedia Tools and Applications* 83.20 (June 1, 2024), pp. 58029–58042. ISSN: 1573-7721. DOI: 10.1007/s11042-023-17167-y. URL: https://doi.org/10.1007/s11042-023-17167-y.

[19] Pegah Gazori, Dadmehr Rahbari, and Mohsen Nickray. “Saving time and cost on the scheduling of fog-based IoT applications using deep reinforcement learning approach”. In: *Future Generation Computer Systems* 110 (Sept. 1, 2020), pp. 1098–1115. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.09.060. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19308702 (visited on 02/15/2025).

[20] Mostafa Ghobaei-Arani and Ali Shahidinejad. “A cost-efficient IoT service placement approach using whale optimization algorithm in fog computing environment”. In: *Expert Systems with Applications* 200 (Aug. 15, 2022), p. 117012. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2022.117012. URL: https://www.sciencedirect.com/science/article/pii/S0957417422004304 (visited on 02/18/2024).

[21] Amirhossein Ghodrati and Shahriar Lotfi. “A Hybrid CS/PSO Algorithm for Global Optimization”. In: *Intelligent Information and Database Systems*. Ed. by Jeng-Shyang Pan, Shyi-Ming Chen, and Ngoc Thanh Nguyen. Berlin, Heidelberg: Springer, 2012, pp. 89–98. ISBN: 978-3-642-28493-9. DOI: 10.1007/978-3-642-28493-9_11. URL: https://link.springer.com/chapter/10.1007/978-3-642-28493-9_11 (visited on 02/15/2025).

[22] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. “Scheduling IoT Applications in Edge and Fog Computing Environments: A Taxonomy and Future Directions”. In: *ACM Computing Surveys* 55.7 (Dec. 15, 2022), 152:1–152:41. ISSN: 0360-0300. DOI: 10.1145/3544836. URL: https://dl.acm.org/doi/10.1145/3544836 (visited on 02/15/2025).

[23] Carlos Guerrero, Isaac Lera, and Carlos Juiz. “Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures”. In: *Future Generation Computer Systems* 97 (Aug. 1, 2019), pp. 131–144. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.02.056. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18325147 (visited on 02/15/2025).

[24] Yan Guo et al. “User allocation-aware edge cloud placement in mobile edge computing”. In: *Software: Practice and Experience* 50.5 (May 2020), pp. 489–502. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.2685. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.2685 (visited on 02/15/2025).

[25] Harshit Gupta et al. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296. ISSN: 1097-024X. DOI: 10.1002/spe.2509. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509 (visited on 02/15/2025).

[26] F. Al-Haidari, M. Sqalli, and K. Salah. “Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources”. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 2. Bristol, UK, Dec. 2013, pp. 256–261. DOI: 10.1109/CloudCom.2013.142. URL: https://ieeexplore.ieee.org/abstract/document/6735431 (visited on 02/15/2025).

[27] Rui Han et al. “EdgeTuner: Fast Scheduling Algorithm Tuning for Dynamic Edge-Cloud Workloads and Resources”. In: *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. ISSN: 2641-9874. London, UK, May 2022, pp. 880–889. DOI: 10.1109/INFOCOM48880.2022.9796792. URL: https://ieeexplore.ieee.org/abstract/document/9796792 (visited on 02/15/2025).

[28] Yoko Hiroshima and Norihisa Komoda. “Parameter optimization for hybrid auto-scaling mechanism”. In: *2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI)*. ISSN: 2471-9269. Budapest, Hungary, Nov. 2016, pp. 000111–000116. DOI: 10.1109/CINTI.2016.7846388. URL: https://ieeexplore.ieee.org/document/7846388 (visited on 02/15/2025).

[29] Xiaobin Hong et al. “An Autonomous Evolutionary Approach to Planning the IoT Services Placement in the Cloud-Fog-IoT Ecosystem”. In: *Journal of Grid Computing* 20.3 (Sept. 14, 2022), p. 32. ISSN: 1572-9184. DOI: 10.1007/s10723-022-09622-1. URL: https://doi.org/10.1007/s10723-022-09622-1 (visited on 02/15/2025).

[30] Jiwei Huang, Yihan Lan, and Minfeng Xu. “A Simulation-Based Approach of QoS-Aware Service Selection in Mobile Edge Computing”. In: *Wireless Communications and Mobile Computing* 2018 (Nov. 1, 2018), p. 5485461. ISSN: 1530-8669. DOI: 10.1155/2018/5485461. URL: https://www.hindawi.com/journals/wcmc/2018/5485461/ (visited on 02/15/2025).

[31] Jiwei Huang, Jingyu Liang, and Sikandar Ali. “A Simulation-Based Optimization Approach for Reliability-Aware Service Composition in Edge Computing”. In: *IEEE Access* 8 (2020). Conference Name: IEEE Access, pp. 50355–50366. ISSN: 2169-

3536. DOI: 10.1109/ACCESS.2020.2979970. URL: https://ieeexplore.ieee.org/abstract/document/9032180 (visited on 02/15/2025).

[32] Yaodong Huang et al. "Mobility-aware Seamless Virtual Function Migration in Deviceless Edge Computing Environments". In: *IEEE Transactions on Mobile Computing* (2023), pp. 1–17. ISSN: 1558-0660. DOI: 10.1109/TMC.2023.3343969. URL: https://ieeexplore.ieee.org/abstract/document/10363648 (visited on 02/15/2025).

[33] Md Muzakkir Hussain et al. "SONG: A Multi-Objective Evolutionary Algorithm for Delay and Energy Aware Facility Location in Vehicular Fog Networks". In: *Sensors* 23.2 (Jan. 2023). Number: 2 Publisher: Multidisciplinary Digital Publishing Institute, p. 667. ISSN: 1424-8220. DOI: 10.3390/s23020667. URL: https://www.mdpi.com/1424-8220/23/2/667 (visited on 02/15/2025).

[34] Md. Muzakkir Hussain and M. M. Sufyan Beg. "CODE-V: Multi-hop computation offloading in Vehicular Fog Computing". In: *Future Generation Computer Systems* 116 (Mar. 1, 2021), pp. 86–102. ISSN: 0167-739X. DOI: 10.1016/j.future.2020.09.039. URL: https://www.sciencedirect.com/science/article/pii/S0167739X20303526 (visited on 02/15/2025).

[35] Mohamed K. Hussein and Mohamed H. Mousa. "Efficient Task Offloading for IoT-Based Applications in Fog Computing Using Ant Colony Optimization". In: *IEEE Access* 8 (2020), pp. 37191–37201. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2975741. URL: https://ieeexplore.ieee.org/document/9006805 (visited on 02/15/2025).

[36] Sundas Iftikhar et al. "HunterPlus: AI based energy-efficient task scheduling for cloud–fog computing environments". In: *Internet of Things* 21 (Apr. 1, 2023), p. 100667. ISSN: 2542-6605. DOI: 10.1016/j.iot.2022.100667. URL: https://www.sciencedirect.com/science/article/pii/S2542660522001482 (visited on 02/15/2025).

[37] Lester James V. Miranda. "PySwarms: a research toolkit for Particle Swarm Optimization in Python". In: *The Journal of Open Source Software* 3.21 (Jan. 10, 2018), p. 433. ISSN: 2475-9066. DOI: 10.21105/joss.00433. URL: http://joss.theoj.org/papers/10.21105/joss.00433 (visited on 02/15/2025).

[38] Artjom Joosen et al. *Serverless Cold Starts and Where to Find Them*. Oct. 8, 2024. DOI: 10.48550/arXiv.2410.06145. URL: http://arxiv.org/abs/2410.06145 (visited on 02/15/2025).

[39] Puneet Kansal, Manoj Kumar, and Om Prakash Verma. "Classification of resource management approaches in fog/edge paradigm and future research prospects: a systematic review". In: *The Journal of Supercomputing* 78.11 (July 1, 2022), pp. 13145–13204. ISSN: 1573-0484. DOI: 10.1007/s11227-022-04338-1. URL: https://doi.org/10.1007/s11227-022-04338-1 (visited on 02/15/2025).

[40] Dervis Karaboga and Bahriye Basturk. “A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm”. In: *Journal of Global Optimization* 39.3 (Nov. 1, 2007), pp. 459–471. ISSN: 1573-2916. DOI: 10.1007/s10898-007-9149-x. URL: https://doi.org/10.1007/s10898-007-9149-x (visited on 02/15/2025).

[41] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4. Nov. 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968. URL: https://ieeexplore.ieee.org/document/488968 (visited on 02/15/2025).

[42] Danylo Khalyeyev, Tomas Bureš, and Petr Hnětynka. “Towards Characterization of Edge-Cloud Continuum”. In: *Software Architecture. ECSA 2022 Tracks and Workshops*. Ed. by Thais Batista et al. Cham: Springer International Publishing, 2023, pp. 215–230. ISBN: 978-3-031-36889-9. DOI: 10.1007/978-3-031-36889-9_16. (Visited on 02/15/2025).

[43] Wazir Zada Khan et al. “Edge computing: A survey”. In: *Future Generation Computer Systems* 97 (Aug. 1, 2019), pp. 219–235. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.02.050. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18319903 (visited on 02/25/2025).

[44] Jonas Krause et al. “7 - A Survey of Swarm Algorithms Applied to Discrete Optimization Problems”. In: *Swarm Intelligence and Bio-Inspired Computation*. Ed. by Xin-She Yang et al. Oxford: Elsevier, Jan. 1, 2013, pp. 169–191. ISBN: 978-0-12-405163-8. DOI: 10.1016/B978-0-12-405163-8.00007-7. URL: https://www.sciencedirect.com/science/article/pii/B9780124051638000077 (visited on 02/15/2025).

[45] Ajay Kumar and Seema Bawa. “A comparative review of meta-heuristic approaches to optimize the SLA violation costs for dynamic execution of cloud services”. In: *Soft Computing* 24 (Mar. 1, 2020). DOI: 10.1007/s00500-019-04155-4. URL: https://link.springer.com/article/10.1007/s00500-019-04155-4 (visited on 02/15/2025).

[46] Dinesh Kumar et al. “A survey on nature-inspired techniques for computation offloading and service placement in emerging edge technologies”. In: *World Wide Web* 25.5 (Sept. 1, 2022), pp. 2049–2107. ISSN: 1573-1413. DOI: 10.1007/s11280-022-01053-y. URL: https://doi.org/10.1007/s11280-022-01053-y (visited on 02/15/2025).

[47] Isaac Lera, Carlos Guerrero, and Carlos Juiz. “Availability-Aware Service Placement Policy in Fog Computing Based on Graph Partitions”. In: *IEEE Internet of Things Journal* 6.2 (Apr. 2019), pp. 3641–3651. ISSN: 2327-4662. DOI: 10.1109/JIOT.2018.2889511. URL: https://ieeexplore.ieee.org/document/8588297 (visited on 02/15/2025).

[48] Chuan Lin, Anyong Qing, and Quanyuan Feng. "A comparative study of crossover in differential evolution". In: *Journal of Heuristics* 17.6 (Dec. 2011), pp. 675–703. ISSN: 1381-1231, 1572-9397. DOI: 10.1007/s10732-010-9151-1. URL: http://link.springer.com/10.1007/s10732-010-9151-1 (visited on 02/15/2025).

[49] Chang Liu et al. "Solving the Multi-Objective Problem of IoT Service Placement in Fog Computing Using Cuckoo Search Algorithm". In: *Neural Processing Letters* 54.3 (June 1, 2022), pp. 1823–1854. ISSN: 1573-773X. DOI: 10.1007/s11063-021-10708-2. URL: https://doi.org/10.1007/s11063-021-10708-2 (visited on 05/10/2024).

[50] Sean Luke. *Essentials of Metaheuristics*. 2. ed. S.l.: Lulu, 2013. 239 pp. ISBN: 978-1-300-54962-8. URL: http://cs.gmu.edu/$%5Csim$sean/book/metaheuristics/ (visited on 02/15/2025).

[51] Sean Luke and AKM Khaled Ahsan Talukder. "Is the meta-EA a viable optimization method?" In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. GECCO '13. New York, NY, USA: Association for Computing Machinery, July 6, 2013, pp. 1533–1540. ISBN: 978-1-4503-1963-8. DOI: 10.1145/2463372.2465806. URL: https://dl.acm.org/doi/10.1145/2463372.2465806 (visited on 02/15/2025).

[52] Quyuan Luo et al. "Resource Scheduling in Edge Computing: A Survey". In: *IEEE Communications Surveys & Tutorials* 23.4 (2021), pp. 2131–2165. ISSN: 1553-877X. DOI: 10.1109/COMST.2021.3106401. URL: https://ieeexplore.ieee.org/document/9519636 (visited on 02/15/2025).

[53] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. "Fog Computing: A Taxonomy, Survey and Future Directions". In: *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*. Ed. by Beniamino Di Martino et al. Singapore: Springer, 2018, pp. 103–130. ISBN: 978-981-10-5861-5. DOI: 10.1007/978-981-10-5861-5_5. URL: https://doi.org/10.1007/978-981-10-5861-5_5 (visited on 02/15/2025).

[54] Peter Mell and Tim Grance. *The NIST Definition of Cloud Computing*. NIST Special Publication (SP) 800-145. National Institute of Standards and Technology, Sept. 28, 2011. DOI: 10.6028/NIST.SP.800-145. URL: https://csrc.nist.gov/pubs/sp/800/145/final (visited on 02/15/2025).

[55] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. 2nd ed. Heidelberg: Springer Berlin, Heidelberg, Sept. 21, 2004. 554 pp. ISBN: 978-3-540-22494-5. URL: https://doi.org/10.1007/978-3-662-07807-5 (visited on 02/15/2025).

[56] Zahra Makki Nayeri, Toktam Ghafarian, and Bahman Javadi. "Application placement in Fog computing with AI approach: Taxonomy and a state of the art survey". In: *Journal of Network and Computer Applications* 185 (July 1, 2021), p. 103078. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2021.103078. URL:

https://www.sciencedirect.com/science/article/pii/S1084804521000989 (visited on 02/15/2025).

[57] Saqib Nazir et al. "Cuckoo Optimization Algorithm Based Job Scheduling Using Cloud and Fog Computing in Smart Grid". In: *Advances in Intelligent Networking and Collaborative Systems*. Ed. by Fatos Xhafa, Leonard Barolli, and Michal Greguš. Cham: Springer International Publishing, 2019, pp. 34–46. ISBN: 978-3-319-98557-2. DOI: 10.1007/978-3-319-98557-2_4. (Visited on 02/15/2025).

[58] Binh Minh Nguyen et al. "Evolutionary Algorithms to Optimize Task Scheduling Problem for the IoT Based Bag-of-Tasks Application in Cloud–Fog Computing Environment". In: *Applied Sciences* 9.9 (Jan. 2019). Publisher: Multidisciplinary Digital Publishing Institute, p. 1730. ISSN: 2076-3417. DOI: 10.3390/app9091730. URL: https://www.mdpi.com/2076-3417/9/9/1730 (visited on 02/15/2025).

[59] *Pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/ (visited on 02/15/2025).

[60] Suraj Pandey et al. "A Particle Swarm Optimization-Based Heuristic for Scheduling Workflow Applications in Cloud Computing Environments". In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. ISSN: 2332-5658. Apr. 2010, pp. 400–407. DOI: 10.1109/AINA.2010.31. URL: https://ieeexplore.ieee.org/abstract/document/5474725 (visited on 02/15/2025).

[61] Milan Patel et al. *Mobile edge computing—a key technology towards 5g*. 11. Sopphia Antipolis, France: ETSI, 2015, pp. 1–16.

[62] Kian Pouresmaeil. "Recommendation for Orchestration Architectures in Serverless Edge Computing". PhD thesis. Vienna, Austria: Technical university of Vienna, 2021. URL: https://doi.org/10.34726/hss.2024.113411..

[63] Wei Qin et al. "MCOTM: Mobility-aware computation offloading and task migration for edge computing in industrial IoT". In: *Future Generation Computer Systems* 151 (Feb. 1, 2024), pp. 232–241. ISSN: 0167-739X. DOI: 10.1016/j.future.2023.10.004. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23003795 (visited on 02/15/2025).

[64] Philipp Raith et al. "An End-to-End Framework for Benchmarking Edge-Cloud Cluster Management Techniques". In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. CA, USA, Sept. 2022, pp. 22–28. DOI: 10.1109/IC2E55432.2022.00010. URL: https://ieeexplore.ieee.org/abstract/document/9946322 (visited on 02/15/2025).

[65] Philipp Raith et al. "faas-sim: A trace-driven simulation framework for serverless edge computing platforms". In: *Software: Practice and Experience* 53.12 (2023), pp. 2327–2361. ISSN: 1097-024X. DOI: 10.1002/spe.3277. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3277 (visited on 02/15/2025).

[66] Philipp Raith et al. “Mobility-Aware Serverless Function Adaptations Across the Edge-Cloud Continuum”. In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. Vancouver, WA, USA: IEEE, Dec. 2022, pp. 123–132. ISBN: 978-1-6654-6087-3. DOI: 10.1109/UCC56403.2022.00023. URL: https://ieeexplore.ieee.org/document/10061786/ (visited on 02/15/2025).

[67] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. “Optimized container scheduling for data-intensive serverless edge computing”. In: *Future Generation Computer Systems* 114 (Jan. 2021), pp. 259–271. ISSN: 0167739X. DOI: 10.1016/j.future.2020.07.017. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X2030399X (visited on 02/15/2025).

[68] Angelina Jane Reyes-Medina, Gregorio Toscano Pulido, and José Gabriel Ramírez-Torres. “A Comparative Study of Neighborhood Topologies for Particle Swarm Optimizers”. In: International Conference on Evolutionary Computation. Vol. 2. SCITEPRESS, Oct. 5, 2009, pp. 152–159. ISBN: 978-989-674-014-6. DOI: 10.5220/0002324801520159. URL: https://www.scitepress.org/PublishedPapers/2009/23248 (visited on 02/15/2025).

[69] Andy M. Reynolds and Mark A. Frye. “Free-Flight Odor Tracking in Drosophila Is Consistent with an Optimal Intermittent Scale-Free Search”. In: *PLOS ONE* 2.4 (Apr. 4, 2007). Publisher: Public Library of Science, e354. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0000354. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0000354 (visited on 02/15/2025).

[70] S. Ronald. “Robust encodings in genetic algorithms: a survey of encoding issues”. In: *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC ’97)*. Apr. 1997, pp. 43–48. DOI: 10.1109/ICEC.1997.592265. URL: https://ieeexplore.ieee.org/abstract/document/592265 (visited on 02/15/2025).

[71] Palash Roy et al. “AI-enabled mobile multimedia service instance placement scheme in mobile edge computing”. In: *Computer Networks* 182 (Dec. 9, 2020), p. 107573. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2020.107573. URL: https://www.sciencedirect.com/science/article/pii/S1389128620312160 (visited on 02/15/2025).

[72] Mahadev Satyanarayanan et al. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 1558-2590. DOI: 10.1109/MPRV.2009.82. URL: https://ieeexplore.ieee.org/document/5280678 (visited on 02/15/2025).

[73] *Serverless Computing – AWS Lambda Pricing – Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/lambda/pricing/ (visited on 02/03/2025).

[74] Olena Skarlat et al. “Optimized IoT service placement in the fog”. In: *Service Oriented Computing and Applications* 11.4 (Dec. 1, 2017), pp. 427–443. ISSN: 1863-2394. DOI: 10.1007/s11761-017-0219-8. URL: https://doi.org/10.1007/s11761-017-0219-8 (visited on 02/15/2025).

[75] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. “EdgeCloudSim: An environment for performance evaluation of Edge Computing systems”. In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. Valencia, Spain, May 2017, pp. 39–44. DOI: 10.1109/FMEC.2017.7946405. URL: https://ieeexplore.ieee.org/document/7946405 (visited on 02/15/2025).

[76] Rainer Storn and Kenneth Price. “Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces”. In: *Journal of Global Optimization* 11.4 (Dec. 1, 1997), pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328. URL: https://doi.org/10.1023/A:1008202821328 (visited on 02/15/2025).

[77] Martin Straesser et al. “Why Is It Not Solved Yet? Challenges for Production-Ready Autoscaling”. In: *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. ICPE ’22. New York, NY, USA: Association for Computing Machinery, Apr. 9, 2022, pp. 105–115. ISBN: 978-1-4503-9143-6. DOI: 10.1145/3489525.3511680. URL: https://dl.acm.org/doi/10.1145/3489525.3511680 (visited on 02/15/2025).

[78] Salman Taherizadeh and Marko Grobelnik. “Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications”. In: *Advances in Engineering Software* 140 (Feb. 1, 2020), p. 102734. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2019.102734. URL: https://www.sciencedirect.com/science/article/pii/S0965997819304375 (visited on 02/15/2025).

[79] Zhiqing Tang et al. “Migration Modeling and Learning Algorithms for Containers in Fog Computing”. In: *IEEE Transactions on Services Computing* 12.5 (Sept. 2019), pp. 712–725. ISSN: 1939-1374. DOI: 10.1109/TSC.2018.2827070. URL: https://ieeexplore.ieee.org/document/8338124 (visited on 02/15/2025).

[80] Mutaz A. B. Al-Tarawneh. “Bi-objective optimization of application placement in fog computing environments”. In: *Journal of Ambient Intelligence and Humanized Computing* 13.1 (Jan. 1, 2022), pp. 445–468. ISSN: 1868-5145. DOI: 10.1007/s12652-021-02910-w. URL: https://doi.org/10.1007/s12652-021-02910-w (visited on 02/15/2025).

[81] László Toka et al. “Machine Learning-Based Scaling Management for Kubernetes Edge Clusters”. In: *IEEE Transactions on Network and Service Management* 18.1 (Mar. 2021), pp. 958–972. ISSN: 1932-4537. DOI: 10.1109/TNSM.2021.3052837. URL: https://ieeexplore.ieee.org/abstract/document/9328525 (visited on 02/15/2025).

[82] Vinita Tomar, Mamta Bansal, and Pooja Singh. “Metaheuristic Algorithms for Optimization: A Brief Review”. In: *Engineering Proceedings* 59.1 (2024). Publisher: Multidisciplinary Digital Publishing Institute, p. 238. ISSN: 2673-4591. DOI: 10.3390/engproc2023059238. URL: https://www.mdpi.com/2673-4591/59/1/238 (visited on 02/15/2025).

[83] Minh-Quang Tran et al. “Task Placement on Fog Computing Made Efficient for IoT Application Provision”. In: *Wireless Communications and Mobile Computing* 2019.1 (Jan. 10, 2019). Publisher: Hindawi, p. 6215454. ISSN: 1530-8669. DOI: 10.1155/2019/6215454. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2019/6215454 (visited on 02/15/2025).

[84] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. “CILP: Co-Simulation-Based Imitation Learner for Dynamic Resource Provisioning in Cloud Computing Environments”. In: *IEEE Transactions on Network and Service Management* 20.4 (Dec. 2023), pp. 4448–4460. ISSN: 1932-4537. DOI: 10.1109/TNSM.2023.3268250. URL: https://ieeexplore.ieee.org/abstract/document/10104137 (visited on 02/15/2025).

[85] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. “GOSH: Task Scheduling Using Deep Surrogate Models in Fog Computing Environments”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (Nov. 2022), pp. 2821–2833. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3136672. URL: https://ieeexplore.ieee.org/document/9656655 (visited on 02/15/2025).

[86] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. “SimTune: bridging the simulator reality gap for resource management in edge-cloud computing”. In: *Scientific Reports* 12.1 (Nov. 10, 2022). Publisher: Nature Publishing Group, p. 19158. ISSN: 2045-2322. DOI: 10.1038/s41598-022-23924-0. URL: https://www.nature.com/articles/s41598-022-23924-0 (visited on 02/15/2025).

[87] Shreshth Tuli et al. “COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog Computing Environments”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (Jan. 2022), pp. 101–116. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3087349. URL: https://ieeexplore.ieee.org/abstract/document/9448450 (visited on 02/15/2025).

[88] Shreshth Tuli et al. “HUNTER: AI based holistic resource management for sustainable cloud computing”. In: *Journal of Systems and Software* 184 (Feb. 1, 2022), p. 111124. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111124. URL: https://www.sciencedirect.com/science/article/pii/S0164121221002211 (visited on 02/15/2025).

[89] S. Walton et al. “Modified cuckoo search: A new gradient free optimisation algorithm”. In: *Chaos, Solitons & Fractals* 44.9 (Sept. 1, 2011), pp. 710–718. ISSN: 0960-0779. DOI: 10.1016/j.chaos.2011.06.004. URL: https://www.sciencedirect.com/science/article/pii/S096007791100107X (visited on 02/15/2025).

[90] Jiadai Wang et al. “Smart Resource Allocation for Mobile Edge Computing: A Deep Reinforcement Learning Approach”. In: *IEEE Transactions on Emerging Topics in Computing* 9.3 (July 2021), pp. 1529–1541. ISSN: 2168-6750. DOI: 10.1109/TETC.2019.2902661. URL: https://ieeexplore.ieee.org/abstract/document/8657791 (visited on 02/15/2025).

[91] Shilin Wen et al. “Fast DRL-based scheduler configuration tuning for reducing tail latency in edge-cloud jobs”. In: *Journal of Cloud Computing* 12.1 (June 17, 2023), p. 90. ISSN: 2192-113X. DOI: 10.1186/s13677-023-00465-z. URL: https://doi.org/10.1186/s13677-023-00465-z (visited on 02/29/2024).

[92] Shilin Wen et al. “K8sSim: A Simulation Tool for Kubernetes Schedulers and Its Applications in Scheduling Algorithm Optimization”. In: *Micromachines* 14.3 (Mar. 2023). Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, p. 651. ISSN: 2072-666X. DOI: 10.3390/mi14030651. URL: https://www.mdpi.com/2072-666X/14/3/651 (visited on 02/15/2025).

[93] Darrell Whitley and Joan Kauth. *GENITOR: A different genetic algorithm, Tech.* Report CS-88-101. Colorado, US: Colorado State University. Department of Computer Science, 1988.

[94] Ye Xia et al. “Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed IoT applications in the fog”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC ’18. New York, NY, USA: Association for Computing Machinery, Apr. 9, 2018, pp. 751–760. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167215. URL: https://dl.acm.org/doi/10.1145/3167132.3167215 (visited on 02/15/2025).

[95] Jie Xu, Lixing Chen, and Shaolei Ren. “Online Learning for Offloading and Autoscaling in Energy Harvesting Mobile Edge Computing”. In: *IEEE Transactions on Cognitive Communications and Networking* 3.3 (Sept. 2017), pp. 361–373. ISSN: 2332-7731. DOI: 10.1109/TCCN.2017.2725277. URL: https://ieeexplore.ieee.org/abstract/document/7973020 (visited on 02/15/2025).

[96] Xin-She Yang, Suash Deb, and Simon Fong. “Accelerated Particle Swarm Optimization and Support Vector Machine for Business Optimization and Applications”. In: *Networked Digital Technologies*. Ed. by Simon Fong. Vol. 136. Series Title: Communications in Computer and Information Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 53–66. ISBN: 978-3-642-22184-2. DOI: 10.1007/978-3-642-22185-9_6. URL: http://link.springer.com/10.1007/978-3-642-22185-9_6 (visited on 02/15/2025).

[97] Xin-She Yang and Suash Deb. "Cuckoo Search via Lévy flights". In: *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*. Coimbatore, India: IEEE, 2009, pp. 210–214. ISBN: 978-1-4244-5053-4. DOI: 10.1109/NABIC.2009.5393690. URL: http://ieeexplore.ieee.org/document/5393690/ (visited on 02/15/2025).

[98] Farkhnada Zafar et al. "Resource Allocation over Cloud-Fog Framework Using BA". In: *Advances in Network-Based Information Systems*. Ed. by Leonard Barolli et al. Cham: Springer International Publishing, 2019, pp. 222–233. ISBN: 978-3-319-98530-5. DOI: 10.1007/978-3-319-98530-5_19. URL: https://doi.org/10.1007/978-3-319-98530-5_19 (visited on 02/15/2025).

[99] Saman Zahoor et al. "Cloud–Fog–Based Smart Grid Model for Efficient Resource Management". In: *Sustainability* 10.6 (June 2018). Publisher: Multidisciplinary Digital Publishing Institute, p. 2079. ISSN: 2071-1050. DOI: 10.3390/su10062079. URL: https://www.mdpi.com/2071-1050/10/6/2079 (visited on 02/15/2025).

[100] Zhi-Hui Zhan et al. "Adaptive Particle Swarm Optimization". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39.6 (Dec. 2009), pp. 1362–1381. ISSN: 1941-0492. DOI: 10.1109/TSMCB.2009.2015956. URL: https://ieeexplore.ieee.org/document/4812104 (visited on 02/15/2025).

[101] Yudong Zhang, Shuihua Wang, and Genlin Ji. "A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications". In: *Mathematical Problems in Engineering* 2015.1 (2015), p. 931256. ISSN: 1563-5147. DOI: 10.1155/2015/931256. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2015/931256 (visited on 02/15/2025).