universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Disseratation / Title of the Doctoral Thesis

„Modeling and Multifaceted Reconfiguration of Cloud-Based Dynamic Routing“

verfasst von / submitted by

Amirali Amiri, M.Sc.(TUM)

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt / field of study as it appears on the student record sheet:	Informatik
Betreut von / Supervisor:	Univ.-Prof. Dr. Uwe Zdun

Acknowledgements

I would like to thank Univ.-Prof. Dr. Uwe Zdun for his supervision in the six years of working on this dissertation. Moreover, I thank Dr. André van Hoorn for his collaborations during the writing of the doctoral thesis.

I dedicate this work to my mother, Fattaneh Navi, who supported me throughout my life.

Danksagungen

Ich bedanke mich bei Univ.-Prof. Dr. Uwe Zdun für seine Betreuung während der sechs Jahre, in denen ich an dieser Dissertation gearbeitet habe. Außerdem danke ich Dr. André van Hoorn für seine Zusammenarbeit während der Erstellung der Doktorarbeit.

Diese Arbeit widme ich meiner Mutter, Fattaneh Navi, die mich mein ganzes Leben lang unterstützt hat.

Abstract

In today's digital age, service- and cloud-based applications have become increasingly dynamic, requiring runtime system adaptations to manage their complex behavior. To meet this need, cloud computing offers an elastic infrastructure that can dynamically adjust resources and scale applications as needed. However, as cloud-based systems become more complex, manual management of these systems becomes increasingly challenging and cost-ineffective.

To ensure that cloud resources are dynamically reconfigured to meet Quality of Service (QoS) requirements, various research studies have focused on different approaches, such as architecture-based reliability modeling, empirical reliability or resilience assessment, architecture-based performance prediction, and performance analysis in cloud-based systems. Additionally, self-adaptive systems have been developed that use Monitor, Analyze, Plan, Execute, and Knowledge (MAKE-K) loops and similar approaches to realize adaptations, while autoscalers and cloud elasticity promise to maintain stable QoS measures even when workload intensity changes.

However, a higher level of abstraction is necessary to make the reconfiguration process of cloud-based systems automatic. This abstraction models the various available technologies and options, which allows us to capture the domain knowledge needed to make informed decisions when choosing optimal reconfiguration solutions. Additionally, empirical research is crucial for supporting the scientific method and creating trust in new technologies and approaches.

In this doctoral thesis, we have designed and performed multiple experiments to model and understand different QoS requirements, including reliability, performance, and system overload. We have modeled these measurements analytically and statistically, then validated our models empirically. Through this empirical research, we have investigated the trade-offs of different QoS metrics and studied how these requirements affect reliability and performance in centralized or distributed systems.

After gathering empirical evidence, we have used our models for multi-criteria optimization analyses, which give the system self-management ability. The system can assess the situation and automatically adapt cloud resources, choosing an optimal reconfiguration solution. To support this approach, we have focused on capturing domain knowledge in the context of QoS requirements and abstracting available technologies to study them at an architectural level of abstraction. We have also provided architectural analysis tools for self-adaptive service- and cloud-based dynamic-routing systems.

Moreover, we have presented a multifaceted reconfiguration of dynamic routing and studied scenarios where components are idle, steady, and transient, as well as the interplay of these scenarios. This dissertation demonstrates the importance of empirical research in creating trust in new technologies and approaches for managing complex cloud-based

dynamic-routing systems. By modeling and understanding different QoS requirements and their trade-offs, we have developed a self-management system that can dynamically adapt cloud resources to ensure stable QoS metrics, even in the face of changing workload intensity.

Kurzfassung

Im heutigen digitalen Zeitalter sind service- und cloudbasierte Anwendungen zunehmend dynamisch und müssen zur Laufzeit angepasst werden, um ihr komplexes Verhalten zu steuern. Zur Erfüllung dieser Anforderungen bietet Cloud Computing eine elastische Infrastruktur, die in der Lage ist, Ressourcen dynamisch anzupassen und Anwendungen nach Bedarf zu skalieren. Mit zunehmender Komplexität der Cloud-Systeme wird die manuelle Verwaltung dieser Systeme jedoch immer schwieriger und kostspieliger.

Um sicherzustellen, dass Cloud-Ressourcen dynamisch rekonfiguriert werden, um die Anforderungen an die Quality of Service (QoS) zu erfüllen, haben sich verschiedene Forschungsstudien auf unterschiedliche Ansätze konzentriert. Diese Studien umfassen architekturbasierte Zuverlässigkeitsmodellierung, empirische Zuverlässigkeits- oder Resilienzbewertung, architekturbasierte Leistungsvorhersage und Leistungsanalyse in Cloud-basierten Systemen. Darüber hinaus wurden selbst-adaptive Systeme entwickelt, die Monitor, Analyze, Plan, Execute und Knowledge (MAKE-K) Schleifen und ähnliche Ansätze verwenden, um Anpassungen zu implementieren, während Auto-Scaler und Cloud-Elastizität versprechen, stabile QoS zu gewährleisten, auch wenn sich die Arbeitslast ändert.

Um den Prozess der Cloud-Systemkonfiguration zu automatisieren, ist jedoch eine höhere Abstraktionsebene erforderlich, um die verschiedenen verfügbaren Technologien und Optionen zu modellieren und das Domänenwissen zu erfassen, das für die Entscheidungsfindung bei der Auswahl optimaler Rekonfigurationslösungen erforderlich ist. Darüber hinaus ist empirische Forschung unerlässlich, um die wissenschaftliche Methode zu unterstützen. Sie schafft Vertrauen in neue Technologien und Ansätze.

Zu diesem Zweck haben wir mehrere Experimente entworfen und durchgeführt, um verschiedene QoS-Anforderungen, einschließlich Zuverlässigkeit, Leistung und Systemüberlastung, zu modellieren und zu verstehen. Wir haben diese Messungen analytisch und statistisch modelliert und unsere Modelle empirisch validiert. Durch diese empirische Forschung haben wir die Kompromisse zwischen verschiedenen QoS-Anforderungen untersucht. Wir haben analysiert, wie diese Anforderungen die Zuverlässigkeit und Leistung des Systems in zentralisierten oder verteilten Systemen beeinflussen.

Nachdem wir die empirische Evidenz gesammelt hatten, haben wir Multi-Kriterien-Optimierungsmodelle eingesetzt, um das System in die Lage zu versetzen, selbstgesteuert zu agieren. Das System kann die Situation bewerten und Cloud-Ressourcen automatisch anpassen, um eine optimale Rekonfigurationslösung auszuwählen. Um diesen Ansatz zu unterstützen, haben wir uns darauf konzentriert zu modellieren, wie Domänenwissen im Kontext von QoS-Anforderungen erfasst werden kann und wie verfügbare Technologien

abstrahiert werden können, um sie auf einer architektonischen Abstraktionsebene zu untersuchen. Darüber hinaus haben wir architektonische Analysewerkzeuge für selbst-adaptive Dienste und Cloud-basierte dynamische Routing-Systeme bereitgestellt.

Nach der Untersuchung verschiedener QoS-Modelle für dynamisches Routing stellen wir eine vielseitige Rekonfiguration für dynamisches Routing vor. Wir untersuchen Szenarien, in denen Komponenten inaktiv, stabil und transient sind, sowie die Interaktion zwischen diesen Szenarien. Diese Dissertation zeigt die Bedeutung empirischer Forschung, um Vertrauen in neue Technologien und Ansätze zu schaffen, um komplexe Cloud-basierte dynamische Routing-Systeme zu verwalten. Durch die Modellierung und das Verständnis verschiedener QoS-Anforderungen und ihrer Trade-offs haben wir ein selbstverwaltendes System entwickelt, das in der Lage ist, Cloud-Ressourcen dynamisch anzupassen, um eine stabile QoS auch bei sich ändernder Auslastung zu gewährleisten.

Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

Nowadays, the dynamic behavior of service- and cloud-based applications often requires runtime system adaptations. Cloud computing provides an elastic infrastructure to manage this behavior of Internet applications. However, cloud-based systems are becoming increasingly complex and it is hard and cost-ineffective to manage them manually. The subject of the thesis concerns the study of *automatic adaptation in the context of cloud resource management*. Dynamic reconfiguration of cloud resources typically considers Quality of Service (QoS) requirements. For example, [90, 54, 27, 24] use architecture-based reliability modeling, and [19, 65, 92] consider empirical reliability or resilience assessment. [86, 81, 69, 77] study architecture-based performance prediction, and [53, 18, 57, 41] perform a performance analysis in the context of cloud-based systems. *Self-adaptive systems* typically use Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) loops [47, 16, 17] and similar approaches to realize adaptations. Similarly, *autoscalers for the cloud* [20, 97] and research on cloud elasticity [42, 37] promise the stable QoS when facing changing workload intensity.

In all this research, an architect must learn the specific framework and define the QoS levels the framework tries to reach. For instance, consider an e-commerce shop that offers discounted products for a specific location during a period. The application must cope with a sudden incoming load increase which needs to be routed to the services residing in the location. An architect can manually decide on the combination of autoscalers and dynamic routers [45], such as an API gateway [79] or an enterprise service bus [26], to accommodate the system demand. However, to make this process automatic so that the system automatically decides on an optimal configuration in a given situation, a higher level of abstraction that models these options should be introduced. Moreover, a mechanism to model the domain knowledge is required to make an informed decision when choosing an optimal reconfiguration solution.

1.1. Motivation

Empirical research, when used to support the scientific method, plays a fundamental role in modern science. Even though empirical research is rather foundational, it is needed to create trust in new technologies and approaches. Unless we understand how certain factors affect tools and methods, the development and use of a particular technology will essentially be a random act. Empirical research represents a key way to move towards well-founded decisions. In our work, as an initial step, we design and perform multiple experiments to model and understand different QoS requirements, e.g., reliability, performance, and system overload. To do so, we analytically and statistically model these

measurements and empirically validate our models.

We investigate our findings and study the trade-offs of these quality requirements. For example, in software architecture, there is a trade-off between system reliability and performance in centralized or distributed systems. In a centralized system, where decisions are made at one *central entity* component, e.g., in an API gateway [79] or a message broker [45], system reliability is higher because there are fewer points of failure. An architect can replicate this routing scheme to ensure high system reliability [40]. However, the centralized component harms the system's performance since all data processing is done by one component. A distributed architecture can increase performance by parallelization but harms the system's reliability by introducing points of failure.

After we have empirical evidence on the trade-offs of different QoS requirements, we can use our models for Multi-Criteria Optimization (MCO) analyses [4], giving the system a self-management ability. The system can automatically assess the situation and adapt the cloud resources. For example, if the incoming load is higher, the system automatically evaluates different options and chooses an optimal reconfiguration solution. This approach is similar to using MAPE-K loops [47, 16, 17]. We focus on the *knowledge* part and model how to capture *domain knowledge* in the context of QoS requirements of cloud resource management. We abstract available technologies to study them at an architectural level of abstraction. Moreover, we provide an architectural analysis tool that considers a multifaceted reconfiguration of self-adaptive service- and cloud-based systems. Additionally, we study whether our reconfiguration concepts can be applied to other domains using machine-learning techniques.

1.2. Research Overview

This section presents our research method, questions, problems, and contributions.

1.2.1. Research Method

We design multiple scientific experiments to validate different QoS models. These models form the basis of multiple MCO analyses to give the system the ability to automatically evaluate different reconfiguration options and choose a final optimal solution. For this purpose, we develop an architectural analysis tool based on design science research [95, 44]. In design science research, first, a research question is posed, then the *develop/evaluate* cycle is continuously repeated until a satisfactory solution for the research question has been obtained. In the course of this process, the research question can be altered or refined. In the first iterations, usually simplifying assumptions are made, which are removed in a stepwise fashion during later iterations.

Design science research produces different outputs: *constructs*, *models*, *methods,* and *instantiations* [91]. While constructs are the conceptual vocabulary of a domain, models are a set of propositions expressing relationships among constructs. Methods are steps to perform a task, and instantiations are operationalized models, constructs, and methods. Design science research is comprised of five steps:

1. **Awareness of problem:** This might result from different sources like earlier research efforts or other disciplines, resulting in a proposal.

2. **Suggestion:** This is "a creative step where new functionality is envisioned based on a novel configuration of either existing or new and existing elements" [91]. This step results in a *tentative design*.

3. **Development:** The tentative design is further developed and implemented.

4. **Evaluation:** The constructed artifact is evaluated according to criteria implicitly (or sometimes explicitly) mentioned in the proposal, e.g., concerning performance.

5. **Conclusion:** In this step, the evaluation results are judged sufficient or insufficient. They are also consolidated and written up. This last step might create additional iterations in the research loop.

We aim to contribute towards a more robust, comprehensive, and evidence-based understanding of architecting self-adaptive service- and cloud-based systems. It is necessary to collect, model, analyze, assess, and understand both the extant practices and the theoretical underpinnings of the service- and cloud-based application domain. Specifically, this requires us to:

1. systematically categorize the extant practices and patterns at an architectural level of abstraction to model and analyze service- and cloud-based systems.

2. establish an automatic approach to assess different quality-of-service requirements in the context of cloud resource management.

3. automatically calculate and provide actionable reconfiguration solutions as part of a feedback loop.

4. provide a prototypical tool to support the architects in applying the actionable reconfiguration solutions found by our approach.

1.2.2. Research Questions

Based on the information derived from the above process, we aim to answer the following research questions (RQ_n) on architecting self-adaptive dynamic routing systems.

***RQ*$_1$** *What elements are required to automatically assess the different quality-of-service requirements in cloud resource management of service- and cloud-based dynamic routing applications?*

***RQ*$_2$** *How to choose the final reconfiguration option as part of a feedback loop to manage cloud resources efficiently, and how well does this reconfiguration solution perform compared to the case when one architecture configuration runs statically?*

***RQ*$_3$** *What is the architecture of a supporting tool that analyses the system quality-of-service requirements and facilitates the reconfiguration of a dynamic routing application using the optimal configuration solution?*

1.2.3. Research Problems

To answer the research questions, we face the following research problems (P_n) addressed by our studies throughout this dissertation.

***P*$_1$** *Lack of reliability models specific to service- and cloud-based dynamic routing*

To answer RQ_1 and RQ_2, we must provide a reliability model specific to dynamic routing. This model allows us to formally assess the reliability issues of a system as a QoS metric. When making architectural design decisions regarding system reconfigurations, we consider this reliability model to assess the QoS requirements.

***P*$_2$** *Lack of performance models specific to service- and cloud-based dynamic routing*

There are many different performance models studied in the literature (see Section 1.3). However, they are mostly generic and are not specific to service-and cloud-based dynamic routing applications. This problem relates to RQ_1 and RQ_2 to assess the QoS requirements and study their trade-offs.

***P*$_3$** *Lack of an approach to automatically adapt the reliability and performance trade-offs*

To answer RQ_2, we must have a formal and automatic approach that performs MCO analyses using the reliability and performance models to adapt the QoS trade-offs in a running dynamic-routing system. This problem also relates to RQ_3 to provide prototypical tool that supports the reconfiguration of routing applications.

***P*$_4$** *Lack of an approach to autoscale components multidimensionally to prevent overload*

This problem relates to RQ_2 because when reconfiguring the routing schema, we must consider that the reconfiguration does not overload the components. Horizontal autoscaling[1] and vertical autoscaling[2] address this issue. However, considering both approaches in one architectural decision-making step regarding dynamic routing must be provided.

***P*$_5$** *Lack of an approach to statefully deplete and reschedule sporadically-active components*

This problem relates to RQ_2 because to manage cloud resources efficiently, we must deplete idle containers statefully and schedule active components instead. When a depleted container becomes active again, there might be a need for container migration to another cloud node. This process must be formalized and studied.

[1] https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler
[2] https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler

***P*$_6$** *Lack of tool support for the multifaceted reconfiguration of dynamic routing applications*

To answer RQ_3, we must provide the prototypical tool support to analyze the system at runtime and facilitate this multifaceted reconfiguration. This tool focuses on the dynamic-routing domain and investigates the interplay of the different reconfiguration views considered in this dissertation.

1.2.4. Research Contributions

This section provides an overview of all contributions of this dissertation linked to their corresponding research problems described in the previous section. In the course of this study, the contributions were published in the cloud and services conferences such as the IEEE World Congress on Services[3] and the International Conference on Service-Oriented Computing[4] as well as in a prominent journal IEEE Transactions on Services Computing[5]. To support replicability, the data, code, and evaluation logs are available as an open-access artifact in the online appendix of this dissertation[6]. Moreover, the research papers relating to each chapter of this doctoral thesis are available in the online appendix under the corresponding chapter. The contributions (C_n) of this work are listed below. A research-overview figure that visualizes the relations between RQ_n, P_n, and C_n are presented in Figure 1.1.

***C*$_1$** *Analytical reliability model of dynamic routing*

This contribution addresses P_1 by modeling component crashes using Bernoulli processes [90] and calculating the request loss in dynamic-routing applications. We empirically validate our analytical reliability model using an extensive experiment of 1200 hours of runtime (see Chapter 2 for details). Moreover, this contribution relates to P_3 since the proposed model is used for reliability and performance trade-offs adaptation.

> **Reference [11]** A. Amiri, U. Zdun, G. Simhandl, and A. van Hoorn. Impact of service- and cloud-based dynamic routing architectures on system reliability. In *International Conference on Service Oriented Computing (ICSOC)*, 2020.
>
> **DOI** 10.1007/978-3-030-65310-1_13

***C*$_2$** *Performance models and trade-offs analysis*

To address P_2, this contribution calculates the round-trip time of requests as a performance model and conducts statistical regression analysis from the data of our experiment. The performance models allow us to precisely analyze the reliability and performance trade-offs of our private infrastructure.

[3] https://conferences.computer.org/services
[4] https://icsoc2022.spilab.es/
[5] https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=4629386
[6] https://zenodo.org/record/7886168 doi:10.5281/zenodo.7886168

Reference [5] A. Amiri, C. Krieger, U. Zdun, and F. Leymann. Dynamic data routing decisions for compliant data handling in service- and cloud-based architectures: A performance analysis. In *IEEE International Conference on Services Computing (SCC)*, 2019.

DOI 10.1109/SCC.2019.00044

Reference [12] A. Amiri, U. Zdun, and A. van Hoorn. Modeling and empirical validation of reliability and performance trade-offs of dynamic routing in service- and cloud-based architectures. In *IEEE Transactions on Services Computing (TSC)*, 2021.

DOI 10.1109/TSC.2021.3098178

Additionally, to address P_2 and to provide a generalized model that applies to other infrastructures, we provide an analytical performance model of dynamic routing. In addition to analyzing our experiment data, we perform a shorter validation experiment on the Google Cloud Platform (GCP)[7]. Moreover, the performance models in this contribution is used for the quality-of-service trade-offs adaptation addressing P_3.

Reference [13] A. Amiri, U. Zdun, and A. van Hoorn. Analytical modeling and empirical validation of performability of service- and cloud-based dynamic routing architecture patterns. In *IEEE Transactions on Services Computing (TSC)*, forthcoming.

C_3 *Automatic adaptation of reliability and performance trade-offs*

This contribution addresses P_3 and details our self-adaptive approach. We use our reliability and performance models to conduct a multi-criteria optimization analysis and adjust the quality-of-service trade-offs automatically. Moreover, this contribution relates to P_6 to provide prototypical tool support for dynamic-routing reconfiguration.

Reference [14] A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Automatic adaptation of reliability and performance tradeoffs in service- and cloud-based dynamic routing architectures. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2021.

DOI 10.1109/QRS54544.2021.00055

Reference [9] A. Amiri and U. Zdun. Tool support for the adaptation of quality of service trade-offs in service- and cloud-based dynamic routing architectures. In *European Conference on Software Architecture (ECSA)*, forthcoming.

C_4 *Multidimensional autoscaling of dynamic routing applications*

To address P_4, this contribution changes the reconfiguration view to focus on each component separately. We model system components as queuing stations [51] and perform an MCO analysis to prevent system overload. We consider the reconfiguration costs

[7] https://cloud.google.com/

an optimization criterion and perform multidimensional autoscaling to prevent system overload. This contribution also relates to P_6 to provide prototypical tool support.

> **Reference [15]** A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Cost-aware multidimensional auto-scaling of service- and cloud-based dynamic routing to prevent system overload. In *IEEE International Conference on Web Services (ICWS)*, 2022.
>
> **DOI** 10.1109/ICWS55610.2022.00063

C_5 *Stateful depletion and rescheduling of idle components*

This contribution relates to P_5 and provides an analytical model and reconfiguration algorithms to deplete and reschedule idle components statefully. We perform a validation experiment on GCP and find an optimal reconfiguration solution. This contribution relates to P_6 as well to study the multifaceted reconfiguration of dynamic routing.

> **Reference [10]** A. Amiri, U. Zdun, and K. Plakidas. Stateful depletion and scheduling of containers on cloud nodes for efficient resource usage. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2022.
>
> **DOI** 10.1109/QRS57517.2022.00056

C_6 *Multifaceted reconfiguration of dynamic routing applications*

We address P_6 and introduce a multifaceted reconfiguration of dynamic routing systems. This contribution studies the interplay of different configuration views studied in this dissertation. Moreover, we provide prototypical tool support regarding this multifaceted reconfiguration.

> **Reference [7]** A. Amiri and U. Zdun. Cost-aware multifaceted reconfiguration of service- and cloud-based dynamic routing applications. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2023.
>
> **DOI** 10.5281/zenodo.7919227
>
> **Reference [8]** A. Amiri and U. Zdun. Smart and Adaptive Routing Architecture: An Internet-of-Things Traffic Manager Based on Artificial Neural Networks. In *IEEE International Conference on Software Services Engineering (SSE)*, 2023.
>
> **DOI** 10.5281/zenodo.7919351

Figure 1.1 provides a complete overview of the research carried out in this doctoral thesis by visualizing the research questions (RQ_n), the research problems (P_n), and the contributions (C_n) with their connections and relationships between each other. Overall, we studied ten research papers that were included in this thesis.

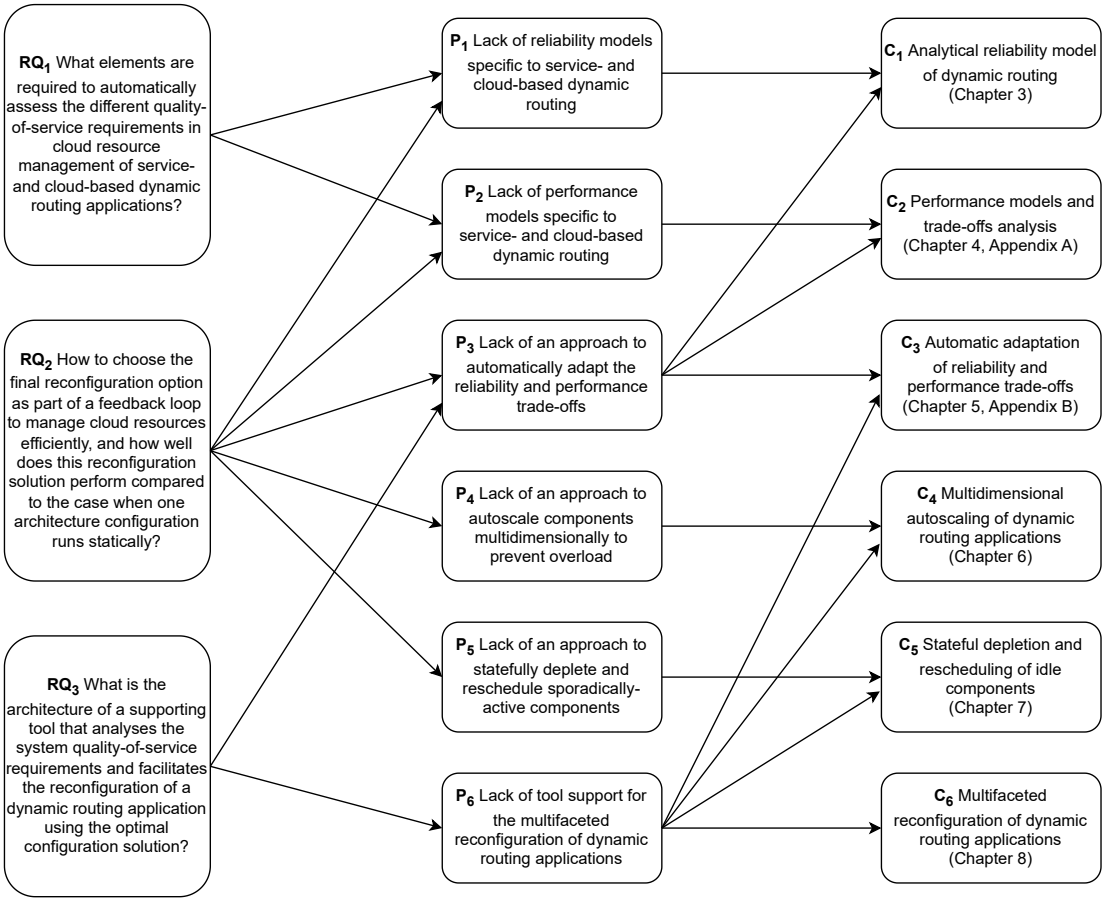


Figure 1.1.: Overview of Research Questions, Problems, and Contributions

1.2.5. Published and Ongoing Research not Included in the Dissertation

We published a research paper and submitted one we did not include in the dissertation. Even though the following research paper is related to our concepts in the sense that it investigates compliance with different rules, its focus is different from that of this dissertation and, consequently, not included in the thesis.

> **Reference [31]** C. Czepa, A. Amiri, E. Ntentos, U. Zdun. Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability. In Software and Systems Modeling, 18 pp. 3331-3371 ISSN 1619-1366 Springer (2019).
>
> **DOI** 10.1007/s10270-019-00721-4

The following paper is a collaboration with Siemens Österreich AG[8]. The focus of this

[8] https://www.siemens.at/

research is on pattern mining for cyber-physical systems, which is not relevant to this dissertation. However, we followed the tool architecture proposed in this paper to provide our prototypical tool support, i.e., RQ_3 and P_6.

> **Reference [6]** A. Amiri, E. Ntentos, U. Zdun, and S. Geiger. Tool Support for Learning Architectural Guidance Models and Pattern Mining from Architectural Design Decision Models. European Conference on Pattern Languages of Programs (EuroPLoP), forthcoming.

1.3. State of the Art

We list the studies common to this dissertation to present state of the art.

1.3.1. Service-Specific Reliability Studies

Our approach, in contrast to many existing architecture-based reliability prediction methods, is focused on a specific category of architectures, namely services-based architectures for dynamic routing. From a practical point of view, reliability in those kinds of architectures has been studied in service and cloud architectures, leading to observations of patterns and best practices [67]. Some works introduce service-specific reliability models. For instance, Wang et al. [94] propose a discrete-time Markov chain model for analyzing system reliability based on constituent services. Grassi and Patella [38] propose an approach for reliability prediction that considers the decentralized and autonomous nature of services. Zheng and Lyu [98] propose an approach that employs past failure data to predict a service's reliability. However, none of these approaches studies and compares major architecture patterns in service and cloud architectures (see Chapter 2 for details). They are rather based on a very generic model with regard to the notion of service. This makes it hard to apply the approaches for prediction when working with specific kinds of architectures, such as those for dynamic routing.

1.3.2. Empirical Reliability or Resilience Assessment

Experiment-based resilience assessment approaches aim to assess a software system's ability to cope with failures, e.g., by injecting faults and observing their effects. Today many software organizations use large-scale experimentation in production systems to assess the reliability of their applications, which is called chaos or resilience engineering [19]. A crucial aspect in resilience assessment of software systems is efficiency [65]. To reduce the number of experiments needed, knowledge about the relationship of resilience patterns, anti-patterns, suitable fault injections, and the system's architecture can be exploited to generate experiments [92]. For example, Pietrantuono et al. [71] propose a new method for adaptive reliability estimation of microservice applications during runtime, which is evaluated experimentally. Our approach differs from these techniques in that our analytical model can be employed to predict the reliability of a software system, whereas key design

decisions, i.e., routers in service- and cloud-based systems, are modeled analytically and assessed empirically.

1.3.3. Architecture-based Reliability Prediction

To predict the reliability of a system and to identify reliability-critical elements of its system architecture, various approaches such as fault tree analysis or methods based on a continuous-time Markov chain have been proposed [54, 27]. Our work can be classified as an architecture-based approach using a Bernoulli process model [90] that is empirically validated (see Chapter 3 for details). Architecture-based approaches, like ours, are often based on the observation that the reliability of a system does not only depend on the reliability of each component but also the probabilistic distribution of the utilization of its components, e.g., formulated as a Markov model. Other approaches allow software engineers to improve the reliability of the software architecture systematically. For example, Brosch et al. [24] suggest an extension of the Palladio component model [21] along with automated transformations into a discrete-time Markov chain [27]. Pitakrat et al. [72] use architectural knowledge to predict how a failure can propagate to other components. They use Bayesian networks to represent conditional dependencies and infer probabilities of failures and their propagation. Our research differs from these approaches in that it focuses specifically on the cloud- and service-based dynamic routing architecture patterns. By focusing on these specific patterns, we can define a more precise model and reach a high level of prediction accuracy.

1.3.4. Architecture-Based Reliability and Performance Prediction

There are studies in this category [90, 30] that employ probabilistic analytical models such as discrete-time Markov chains [27] and (layered) queueing networks [86], or high-level architectural models such as profile-extended Unified Modeling Language (UML) [69] or Palladio [77, 24] models, which are simulated or transformed into analytical models. The approaches are based on the observation that the reliability and performance of a system depend on the reliability and performance of each component, along with the interplay between them. Moreover, Sharma and Trivedi [81] present an architecture-based unified hierarchical model for software reliability, performance, security, and cache behavior prediction. Architecture-based MCO [4] builds on top of these prediction approaches and the application of architectural tactics to search for (pareto) optimal architectural candidates. Example MCO approaches supporting reliability and performance are ArcheOpterix [3], PerOpteryx [25], and SQuAT [75]. Like our study, these works support architectural design. In contrast, our work gives extensive empirical evidence of our architecture-based reliability and performance models. These empirically-validated models are then used in our MCO analysis, where we provide tool support for further experimentation.

1.3.5. Architecture-Based Performance Analysis

Spitznagel and Garlan [86] present a general architecture-based model for performance analysis based on queueing network theory. Petriu et al. [69] present an architecture-based performance analysis that builds layered-queuing-networks performance models from a UML description of the high-level architecture of a system. The Palladio component model [21, 77] allows precise component modeling with relevant factors for performance properties and contains a simulation framework for performance prediction. Like our research, those works focus on supporting architectural design or decision-making. In contrast to our work, they do not focus on specific kinds of architectures or architectural patterns. Those models offer more generality at the expense of the high accuracy with which we characterize the three architecture patterns analyzed in our work.

1.3.6. Performance Analysis: Enterprise Service Buses

There are several existing studies comparing the performance of Enterprise Service Buses (ESB). This is related to our research because ESBs provide a means for the content-based routing of messages. Sanjay et al. [2] evaluate the performance of the three open-source ESBs, i.e., WSO2 [48], Mule [35], and Service Mix [74]. The performance is measured based on mean response time and throughput for proxying, content-based routing, and data mediation. However, the test scenarios only consider client communications and a single web service. Shezi et al. [83] provide a performance evaluation of different ESBs in a more complex scenario in which multiple services are composed to achieve a certain business objective. As a test case, a service orchestration scenario is simulated, in which a consumer consults several banking services to find the best loan quote. In contrast to these studies, our work considers multiple routing architectures. Moreover, we investigate communication paths that involve the composition of multiple services and routing decisions.

1.3.7. Performance Analysis: Microservice- and Container-Based Systems

Different studies evaluate the network performance of container-based applications. This is related to our work, as we analyzed the performance of containerized services. For example, Kratzke [53] evaluates the performance impact of Docker containers and software-defined networks in distributed cloud-based systems using HTTP-based communication. The performance is measured by means of the data transfer rate of m byte-long messages. A similar work is presented by Bankston et al. [18] to explore the network performance and system impact of different container networks on public clouds from Amazon Web Services, Microsoft Azure, and Google Cloud Platform. Another kind of related studies in a wider sense, compares different service architectures. For instance, Lloyd et al. [57] compare different states of serverless infrastructure and their influence on microservice performance. Khazaei et al. [52] study the efficiency of provisioning microservices. All these studies are related to our research as they also improve the state of (micro)service performance engineering. Our study contributes new data on common architectures for

evaluating dynamic routing rules, which has not been examined before. The literature has produced general microservice performance engineering challenges and directions (see e.g., [41]). Studies like ours and those mentioned above address some of the microservice performance engineering challenges identified in the literature.

Additionally, Vandikas et al. [93] conducted a performance analysis of their IoT framework to evaluate its behavior under heavy load produced by different amounts of producers and consumers. The main purpose of the framework is to allow producers, such as sensors, to publish data streams to which multiple interested consumers, e.g., external applications, can subscribe. This publish-subscribe functionality is realized by a central message broker implemented with RabbitMQ. In contrast to our work, dynamic data routing is not considered in this article. Moreover, the performance evaluation of the framework focuses only on a single-machine deployment, which may have led to results that are not easily generalizable to cloud-based deployments.

1.3.8. Studies on Performability of Systems

Performability considers the effects of structural changes in a system, e.g., when there are component crashes (impeded reliability), on the overall performance of the system [85]. Ahamad and Ratneshwer [1] provide a review on the performability of Safety-Critical Systems (SCS). They study the available approaches and the metrics to evaluate the performability of SCS. Moreover, they define the challenges of performance and reliability in studying the SCS. This study is related to our work as it presents state of the art in performability studies. However, in contrast to our work, it does not provide an analytical model of performance and reliability and its empirical validation to improve state of the art. Mo et al. [62] study the performability analysis of multi-state sliding window systems. Like our study, they propose an analytical approach based on multivalued decision diagrams. They analyze this analytical approach in multiple case studies. Lisnianski et al. [55] present a Markov multi-state model for large-scale and highly responsive distributed systems. Similar to our work, they provide an analytical performability model and present a short-term analysis to prevent performance and reliability decreases.

A particular related work is [89], in which Torquato et al. study the migration of virtual machines of a cloud-based system in the presence of workload. Like our work, they provide a modeling framework to support virtualized environment management decisions regarding the performability of a system. A particularly-recent work is [34], in which Di Mauro et al. study containerized network applications. They use queuing network theory and consider each container as a queuing node. Like our research, they performed a cloud-based experiment to evaluate their performability model. In contrast to the above studies, our research is specific to dynamic routing and provides reliability and performance models. Unlike our research, none of the above works provide extensive empirical data supporting the accuracy of their proposed models. To the best of our knowledge, this has not been considered in the literature with regard to dynamic-routing architecture patterns.

1.3.9. Self-Adaptive Systems

Our approach is related to self-adaptive systems, which typically use MAPE-K loops [47, 16, 17] and similar approaches to realize adaptations. Our proposed architecture uses a similar structure as the MAPE-K loop and extends such approaches with support specific to the service- and cloud-based dynamic routing architectures. Similarly, works on cloud elasticity [42, 37] are related to our research. In contrast to the existing related work, the major contribution of our approach is that we focus on specific architecture patterns for dynamic routing and consider reliability and performance trade-offs. By focusing on these specific patterns and possible runtime self-adaptations, we can define a targeted model along with a specific reconfiguration algorithm and preference functions to perform MCO analyses, which would be hard in the generic case.

Moreover, research on efficient resource provisioning, e.g., [52, 29] are related to our work. Our study extends these approaches by analytically modeling the depletion of idle containers as a reconfiguration measure. Similarly, autoscalers for the cloud [20, 97], which promise stable quality-of-service and cost minimization when facing changing workload intensity, are related work. Multidimensional autoscalers have been studied in the literature for resource provisioning. AutoMAP [22] uses response time triggers to provision resources. AutoMAP finds optimal resources using Virtual Machine (VM) image sizes to support cost efficiency. Nguyen et al. [66] suggest a forecasting model to predict CPU demand and use these predictions to start new machines before load peak to increase performance. CloudScale [82] supports scaling CPU and memory resources when local scaling is possible. Otherwise, it migrates VMs to prevent overloaded hosts. Our work differs from these studies because they consider auto-scaling at the VM level and configure the resources. We proposed an approach that works at the container level by depleting and rescheduling containers on cloud nodes.

1.3.10. Container Scheduling

There is a rich literature on container scheduling in a more general sense. For example, Stratus [28] is a dynamically-allocating cluster scheduler that orchestrates batch job execution on clusters of VM instances of public Infrastructure as a Service (IaaS) platforms. Kubernetes container scheduling strategy [60] offers a novel container scheduling strategy for Kubernetes. Kaewkasi and Chuenmuneewong [50] use ant-colony-optimization methods to implement a new scheduler for Docker, whereas Liu et al. [56] provide a new container scheduling approach based on multi-objective optimization. Cérin et al. [32] introduce a new Docker Swarm scheduler that uses service level agreement information to provision a container that must execute the service based on a dynamic computation of available resources. Sureshkumar and Rajesh [88], in contrast to those other approaches, use load scheduling to optimize container usage. Our study differs from these works because it is not specific to container scheduling technology. Our approach tackles containers' stateful depletion and rescheduling to cloud nodes from a higher level of abstraction that can be used with different container orchestration technologies.

1.4. Thesis Structure

The remainder of this thesis is structured as follows: Chapter 2 provides the overview of our proposed Adaptive Dynamic Routers architecture and the common details of our scientific experiments. Chapter 3 presents our reliability model of dynamic routing and its empirical validation as a QoS metric. Chapter 4 introduces our performance models. Firstly, this chapter studies a statistical performance model specific to our private infrastructure. This specificity allows us to perform a detailed analysis of reliability and performance trade-offs discussed in this section. Secondly, this chapter gives an analytical model of performance for dynamic routing to support generalizability to other applications outside of our infrastructure. Chapter 5 presents the details of our approach regarding the automatic adaptation of reliability and performance trade-offs. Moreover, we present prototypical tool support. Chapter 6 provides a different reconfiguration view, i.e., the focus is placed on one component. In this chapter, we model components as queuing stations [51] and present a novel multidimensional autoscaler to prevent system overload. Chapter 7 focuses on stateful depletion and rescheduling of sporadically-active components and presents an experiment to validate our self-adaptive architecture empirically. Chapter 8 presents a multifaceted reconfiguration of dynamic routing and considers the interplay of the different reconfiguration views. All chapters contain a conclusions section; therefore, Chapter 9 summarizes the contributions, presents the planned future work, and concludes the dissertation.

2. Approach Overview

In this chapter, we present the overview of our approach followed in the remainder of this dissertation. Firstly, we provide a background on the dynamic routing architecture patterns. Secondly, we propose a novel Adaptive Dynamic Routers architecture that automatically changes the routing schema based on the monitored data at runtime. Finally, we present the details of the validation experiments that we follow throughout this thesis.

2.1. Background on Dynamic Routing Patterns

There are many different service- and cloud-based architectures that use or enable dynamic request routing. We study three of the widely-used architectural patterns, i.e., the *Central*

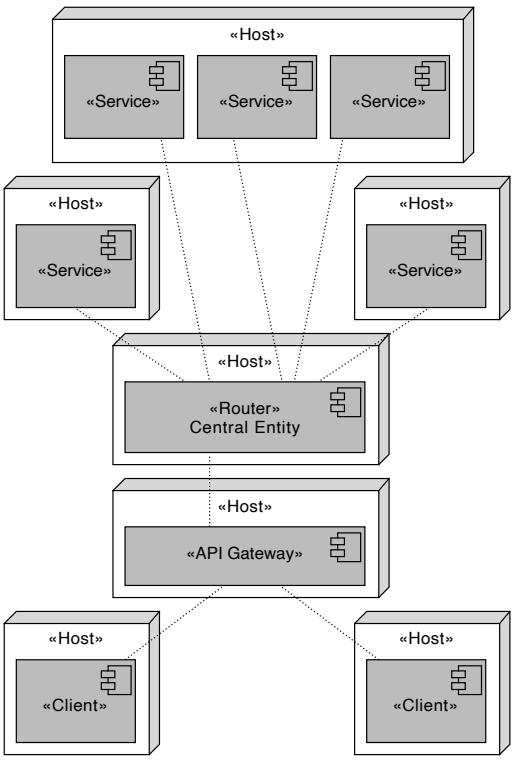


Figure 2.1.: Central Entity Architecture

Entity, the *Sidecar-based Architecture*, and the *Dynamic Routers*.

2.1.1. Central Entity

In a Central Entity (CE) architecture, the central entity manages all request flow decisions. Figure 2.1 on the previous page shows an example configuration of CE. One benefit of this architecture is that it is easy to manage, understand, and change since all control logic regarding request flows is implemented in one component. However, this introduces the drawback that the design of the internals of the central entity component is a complex task. Another advantage is that in an application consisting of stateful request flow sequences, the state does not need to be passed between various distributed components. Nonetheless, services need to call back to the central entity component to fetch the saved state of prior stages to proceed with the next step in a request flow sequence. CE can be implemented using an API gateway [79], an event store or an event streaming platform [79], or any kind of central service bus [26]. Note that it is not required that the central entity component is always deployed on an exclusive host, as shown in Figure 2.1.

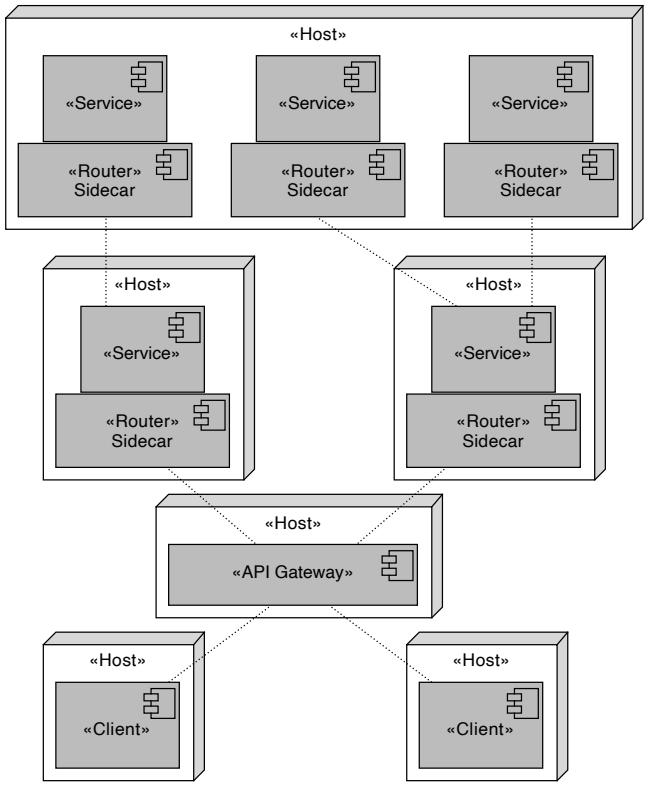


Figure 2.2.: Sidecar-Based Architecture

2.1.2. Sidecar-based Architecture

The Sidecar-based Architecture (SA) is presented in Figure 2.2. In contrast to the central entity architecture, the control logic is distributed and placed in so-called sidecars [49, 36], which are attached to the services. Sidecars offer a separation of concerns since the control logic regarding request flow is implemented in a different component than the service. However, sidecars are tightly coupled with their directly-linked services. The SA architecture offers benefits whenever decisions need to be made structurally close to the service logic. One advantage of this architecture is that, compared to the central entity service, it is usually easier to implement sidecars since they require less complex logic to control the request flow of their connected services. Nonetheless, it is not always possible to add sidecars, e.g., when services are off-the-shelf products. Sidecars are almost always implemented on the same host as their directly-linked services.

2.1.3. Dynamic Routers

Dynamic Routers (DR) [45] can be seen as a hybrid of the two extremes, i.e., between the centralized CE and the fully distributed SA. Figure 2.3 on the next page shows a specific DR configuration. One benefit of this architecture is that dynamic routers can use local information regarding request routing amongst their connected services. For instance, if a set of services are dependent on one another as steps of processing a request, DR can be used to facilitate the routing. Nonetheless, dynamic routers introduce an implementation overhead regarding data structures, control logic, management, deployment, etc., since they are usually distributed on multiple hosts. We use the common term *router* for all request flow control logic components, i.e., the central entity in CE, sidecars in SA, and dynamic routers in DR. This concept is further explained below.

2.2. Adaptive Dynamic Routers Architecture

The CE, DR, and SA architecture patterns are implemented based on very different concepts, including API gateways [79], such as NGINX[1] or Kong[2], enterprise service buses [26], message brokers [45], or sidecars [61, 49, 36] such as Envoy[3]. Essentially they all route the incoming requests dynamically. We propose a new approach that realizes all three architectural patterns, as explained below. We hypothesize that a dynamic self-adaptation between the three architectures is beneficial over any fixed architecture selections. If a traffic and load change occurs, our approach can self-adapt the degrees to which more central or distributed routing is used dynamically. This self-adaptation is performed to optimize the impacts on quality of service trade-offs, e.g., performance and reliability.

We suggest that the trade-offs adaptation can be automated using multi-criteria optimization analyses [4]. This allows us to engineer our novel self-adaptive architecture

[1] https://www.nginx.com
[2] https://konghq.com/kong/
[3] https://www.envoyproxy.io/

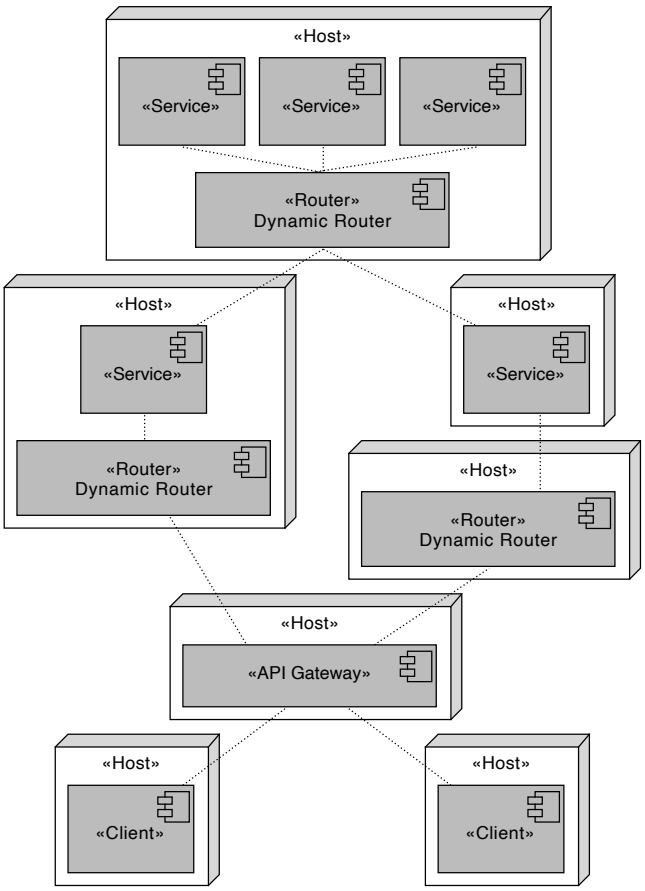


Figure 2.3.: Dynamic Routers Architecture

approach that we call the *Adaptive Dynamic Routers (ADR)* architecture. ADR is based on Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) loops [47, 16, 17] and dynamically adapts between the architecture patterns on-the-fly. As mentioned, we define a concept called *router* and abstract all the controlling logic components, i.e., the central entity service, the dynamic routers, and the sidecars, under the router. This high-level router abstraction can be used to reconfigure the routing architecture dynamically. That is, we can change between the three architectures moving from a centralized approach with one router to a distributed system with more routers (or vice versa) to adapt based on the need of an application.

Let us clarify the difference between the DR and the newly-introduced ADR architectures. DR is a fixed architecture that typically does not change while a system runs. If it changes in reality, a redeployment has to happen, manually designed and often manually deployed by system engineers. In our ADR approach, the system dynamically switches based on the results of optimization analyses that can be triggered, for instance, in

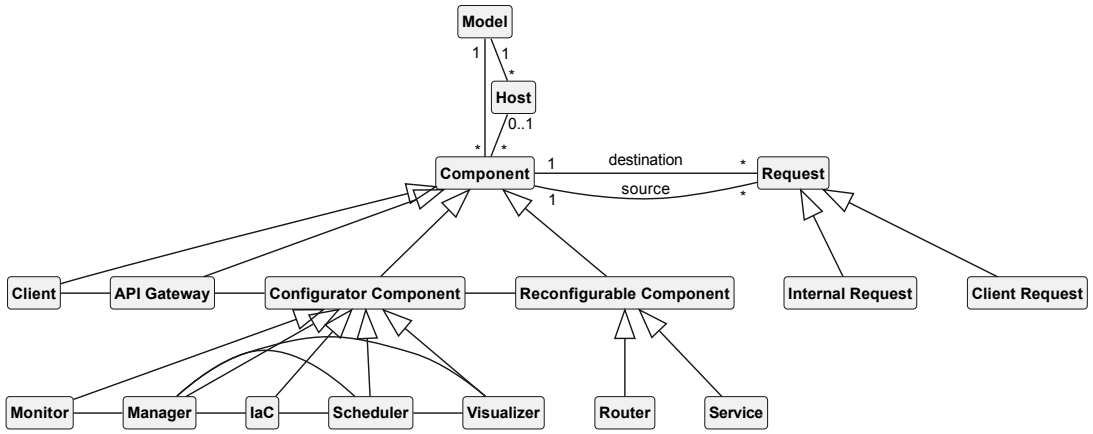


Figure 2.4.: Metamodel of the Adaptive Architecture

certain time intervals or whenever certain metrics change. For example, when round-trip performance degrades, requests fail, incoming load changes, or a different route containing more services is used. In contrast to all three fixed architecture patterns, ADR is adaptive and changes at runtime.

2.2.1. Metamodel

Figure 2.4 presents the metamodel of our architecture. A *Model* describes multiple *Hosts* and *Components*. Each *Component* is deployed on (up to) one *Host* at each point in time, which is any execution environment for these components, either physical or virtual. *Request* models the request flow, linking a source and a destination component. There are several different component types. *Clients* send *Client Requests* to *API Gateways*. To process these requests, *API Gateways* send *Internal Requests* to *Routers* and *Services*.

Routers and *Services* are both *Reconfigurable Components*, i.e., they are the adaptation targets. The *Configurator Components* perform the reconfiguration. *Monitor* observes *Reconfigurable Components* and the requests that pass the *API Gateways*. *Manager* manages the control flow of the reconfiguration by calling the *Infrastructure as Code (IaC)* to update the infrastructure or the *Scheduler* to reschedule the containers.

2.2.2. Example of a Routing Configuration

Figure 2.5 presents a component diagram of a sample configuration, in which dashed lines represent the data flow and solid lines the reconfiguration control flow of an application. As shown, clients access the system via an API gateway that publishes monitoring data to the Quality of Service (QoS) monitor. The configuration manager observes the monitoring data and triggers a reconfiguration. Moreover, the manager can communicate with the visualizer component to visualize the current architecture configuration. The manager calls the IaC component if infrastructure changes are needed, which reconfigures the infrastructure. IaC can also trigger the scheduler to reschedule the containers.

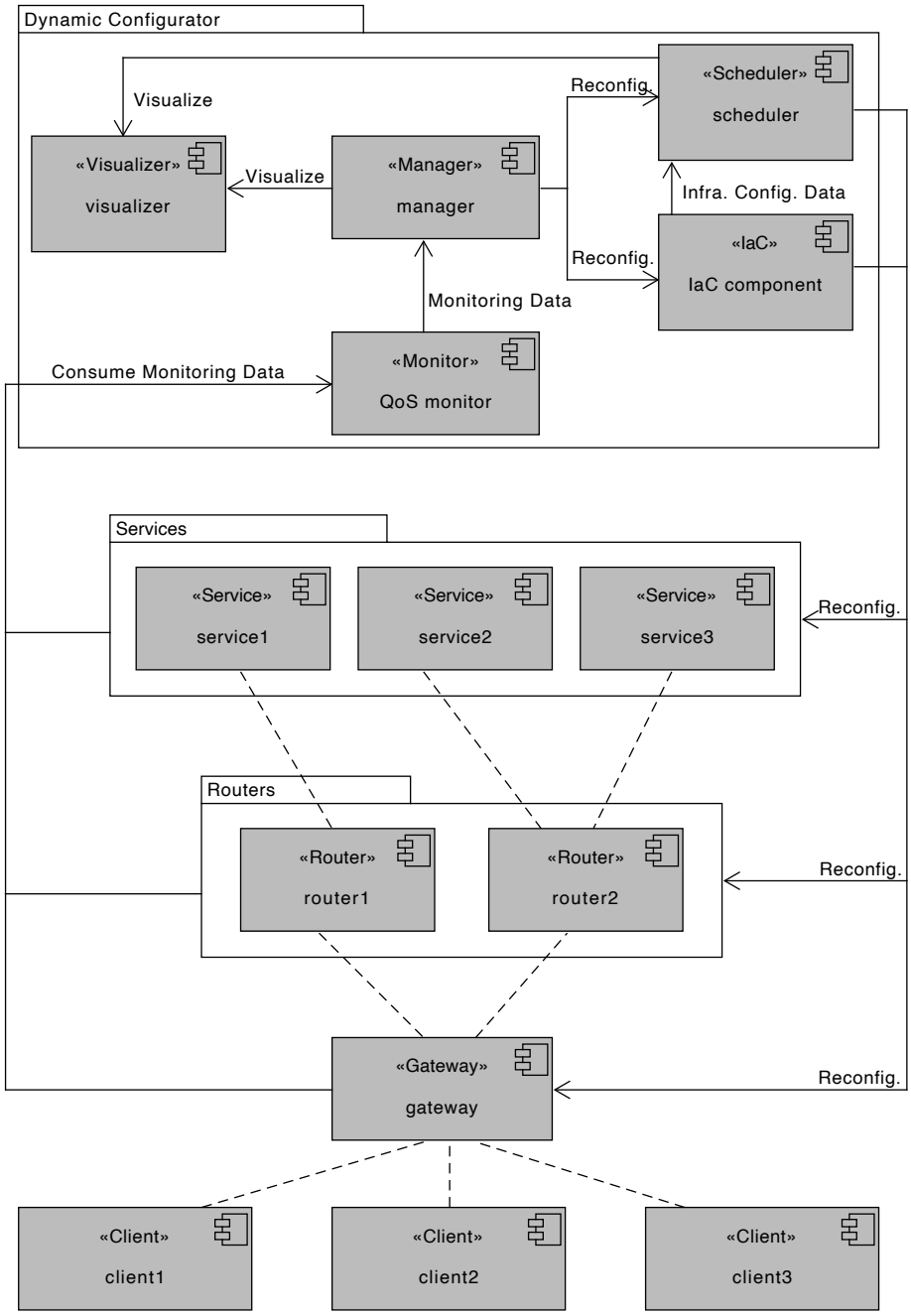


Figure 2.5.: Component Diagram of an Example Configuration (dashed lines represent the data flow and solid lines the reconfiguration control flow of an application.)

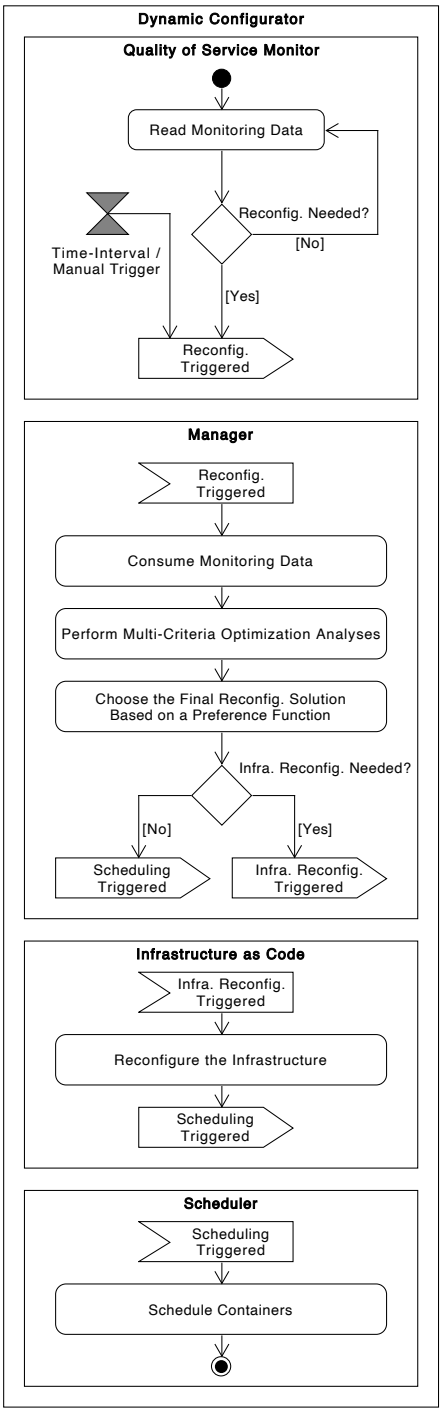


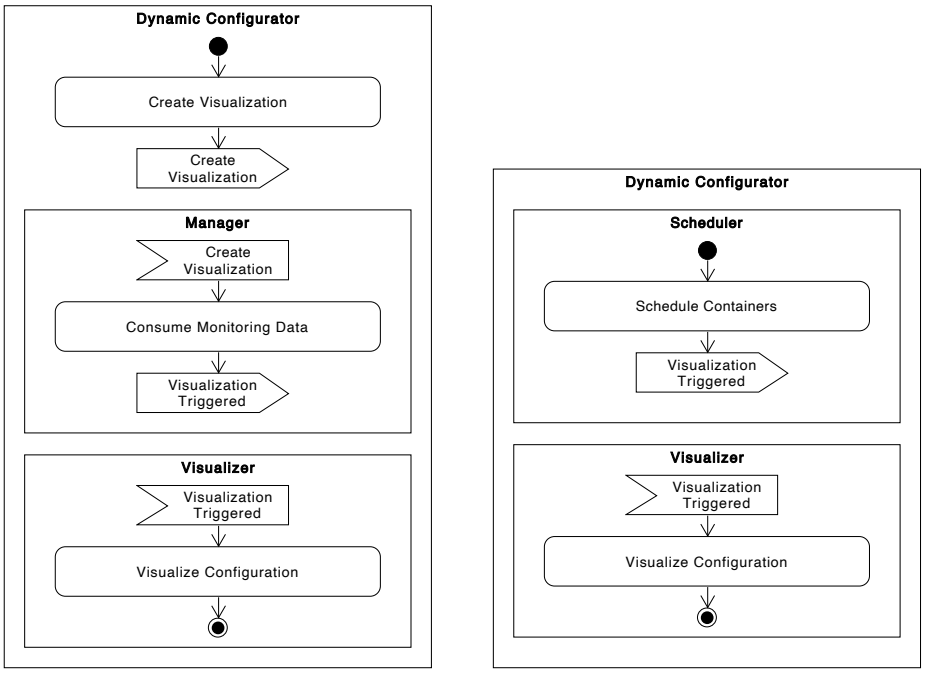
Figure 2.6.: Reconfiguration Activities of the Dynamic Configurator

Alternatively, if there is no need for infrastructure reconfiguration, the manager directly triggers the scheduler. After a reconfiguration, the scheduler can call the visualizer component to visualize the reconfiguration.

2.2.3. Reconfiguration Activities of the Dynamic Configurator

Figure 2.6 shows the reconfiguration activities of the dynamic configurator. The QoS monitor reads monitoring data and checks for reconfiguration, e.g., when degradation of reliability and performance metrics are observed. Moreover, the reconfiguration can be triggered periodically or manually by an architect. When a reconfiguration is triggered, the reconfiguration manager consumes the monitoring data, performs multi-criteria optimization analyses [4], and chooses a final reconfiguration solution. Based on this analysis, either the IaC component is triggered to reconfigure the infrastructure or the scheduler reschedules the containers. These activities are further explained in this section. As mentioned, our architecture is based on MAPE-K loops [47, 16, 17]. The QoS monitor implements the *monitor* and *analyze* stages, the manager develops the *plan* step, and the IaC component and the scheduler realize the *execute* step. We use our models as *knowledge*.

Figure 2.7 shows the visualization activities of the dynamic configurator. The visualizer



(a) Initial Visualization (b) Visualization after Reconfiguration

Figure 2.7.: Visualization Activities of the Dynamic Configurator

is called upon either by the manager or the scheduler in two different scenarios. The manager triggers the visualizer when a user requests to create a visualization of the current architecture configuration as shown by Figure 2.7a. The scheduler calls the visualizer after a reconfiguration is performed and containers are rescheduled. This visualization is created to inform the user of the latest changes in the architecture configuration as shown by Figure 2.7b.

2.3. Details of our Scientific Experiments

In this thesis, we introduce multiple models of quality-of-service attributes. These models are used in our proposed ADR architecture when making architectural design decisions (further explained in the remainder of the dissertation). The goal of our multiple experiments is to validate these models empirically. We provide the details of our main scientific experiments in this chapter to avoid repetition. These experiments are published in multiple studies[13, 12, 11] and include a runtime of 1200 hours on our private cloud infrastructure as well cross-validation on a public cloud infrastructure. Moreover, we performed shorter empirical validations for different chapters of this dissertation, details of which we describe in their respective chapters.

2.3.1. Example ADR Model Instance

We use the concepts of our metamodel to instantiate an example model. Figure 2.8 shows an ADR model instance with three routers and six services. This figure is repeated in multiple chapters of this thesis, where different model elements must be illustrated. The instantiated components send *Internal Requests* amongst one another to complete the processing of the one *Client Request* (see Figure 2.4).

2.3.2. Technical Details

This section provides the technical details of our validation experiments.

Private Cloud Infrastructure

We used a private cloud with three physical nodes, each having two identical CPUs. Two cloud nodes host Intel®Xeon®E5-2680 v4 @2.40GHz[4] and the other one hosts the same processor family but version v3 @2.50GHz. The v4 and v3 versions have 14 and 12 cores for each CPU and two physical threads per core (in total, 56 and 48 threads). On top of the cloud nodes, we installed Virtual Machines (VMs), each using VMware ESXi version 6.7.0 u2 hypervisor with eight CPU cores and 60 GB system memory. All VMs run Ubuntu Server 18.04.01 LTS[5]. Docker[6] containerization is used to run the cloud services

[4]https://www.intel.com/content/www/us/en/homepage.html
[5]https://www.ubuntu.com
[6]https://www.docker.com

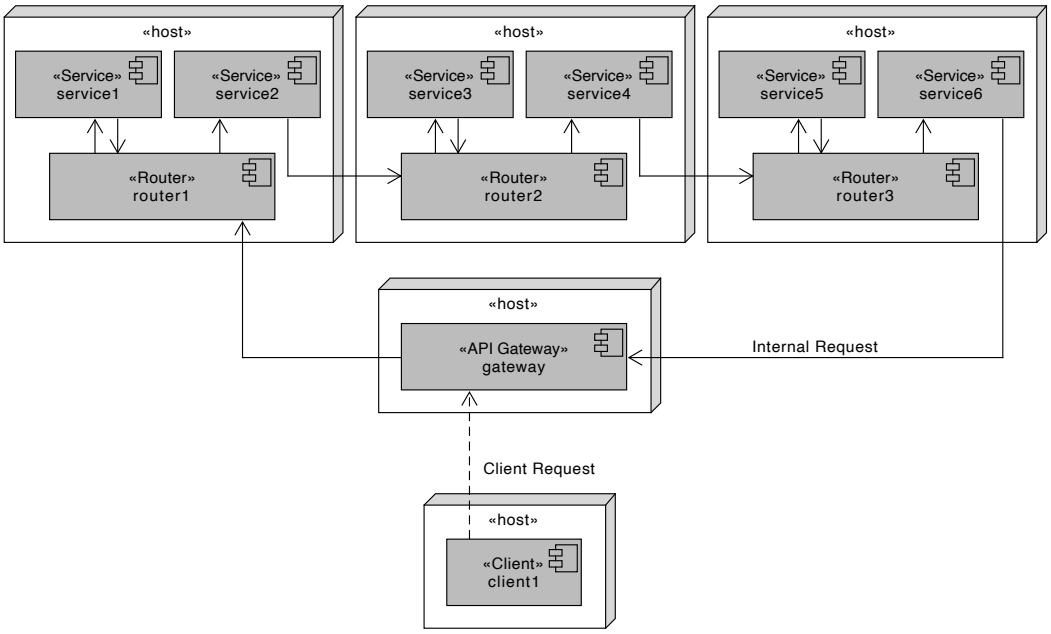


Figure 2.8.: Example Configuration of Dynamic Routing Applications (solid arrows show the incoming requests of components.)

implemented in Node.js[7].

Public Cloud Infrastructure

We use our private cloud to have control over the infrastructure and have repeatable experiment runs. On a public cloud, other factors can influence the results, such as the parallel workload of other applications or the physical distance of the nodes. To show that our approach can be used on other infrastructures as well, we empirically validate our results also on Google Cloud Platform (GCP)[8]. The details of these validation experiments are reported in each chapter separately.

Load Generation

We used five desktop computers to generate load, each hosting an Intel®Core™i3-2120T CPU @2.60GHz with two cores and two physical threads per core. All desktop computers have 8 GB of system memory and run Ubuntu 18.10. They generate load using Apache JMeter[9] that sends Hypertext Transfer Protocol (HTTP) version 1.1[10] requests to the

[7] https://nodejs.org/en/
[8] https://cloud.google.com
[9] https://jmeter.apache.org
[10] https://tools.ietf.org/html/rfc7230

cloud nodes.

2.3.3. Architecture Configurations

Any application that has a request flow can be modeled using our proposed metamodel. We used a few sample architecture configurations to calculate the accuracy of our models. These configurations follow the convention for the request flow shown by the example model in Figure 2.8. All clients send client requests to the API gateway. Internal requests are sent one by one from routers to services and vice versa. Also, for the sake of simplicity, we label the services and the routers incrementally from 1 and make the internal requests go through all of them linearly. Moreover, we distribute services equally among routers.

As in the example model, we utilized one virtual machine exclusively, with only one Docker container, to run the API gateway. We distributed the cloud services, each on a separate container, amongst three VMs. The distribution is so that all virtual machines have the same number of services (with a maximum difference of one service). However, the placement of routers on hosts is different from that of the example model. For centralized routing CE architecture, we placed the router in a Docker container exclusively on one VM. For distributed DR architecture, we used three exclusive VMs, each with only one container for the routers. Finally, for the sidecar architecture pattern [49], we placed each sidecar in a separate container on the same VM, where its directly-linked service resides.

2.3.4. Experiment Cases

Let cf be the incoming call frequency. We selected cf based on a study of related works as 10, 25, 50, and 100 requests per second (r/s). In many related studies (see, e.g., [33, 87]), 100 r/s (or even lower numbers) are chosen. Consequently, we have chosen this number as our highest bound and selected different portions to study its effects.

$$cf \in \{10, 25, 50, 100\}\ r/s \tag{2.1}$$

Let n_{serv} be the number of services in an ADR application. Based on our experience and a survey on existing cloud applications in the literature and industry [5, 12], the number of cloud services directly dependent on each other in a call sequence is usually rather low. As a result, we chose 3, 5, and 10 as values for the number of services in the call sequence:

$$n_{serv} \in \{3, 5, 10\} \tag{2.2}$$

As mentioned, we study three representative architecture configurations. Let n_{rout} be the number of routers in a dynamic routing application. In a centralized routing schema [79, 26], only one router processes all incoming requests, i.e., $n_{rout} = 1$. On the other hand, in a completely distributed routing pattern, such as the sidecar pattern [49], there is one router per each service, i.e., $n_{rout} = n_{serv}$. Additionally, we study a combination of these two extremes, i.e., the dynamic router pattern [45], where several routers process the requests, e.g., one router per each geographical region. For this case, we chose a case of

three routers, i.e., $n_{rout} = 3$, each deployed on a separate VM that processes the requests of services on the same machine.

$$n_{rout} \in \{1, 3, n_{serv}\} \tag{2.3}$$

In this doctoral thesis, we study component crashes and their effects on quality-of-service measures (see Chapter 3 for more details). Let CP_c be the crash probability, CI the crash interval, d_c the downtime of each component after it crashes, T the observed system time, and n_{crash} the number of crash tests per T. In our experiment, we simulated a node crash by generating a random number for each cloud component, i.e., the services and routers. If the generated random number for a component was below its crash probability CP_c, we stopped the component's Docker container and started it again after a time interval of $d = 3$ seconds. We chose $T = 600$ seconds, during which we checked for a crash for all components, i.e., routers and services, simultaneously every $CI = 15$ seconds; therefore, $n_{crash} = 40$.

$$d = 3\,s \tag{2.4}$$
$$T = 600\,s \tag{2.5}$$
$$CI = 15 \tag{2.6}$$
$$n_{crash} = 40 \tag{2.7}$$

Each component had a uniform crash probability of 0.5% each time we checked for a crash, as mathematically expressed by Equation (2.8). Note that this crash probability is much higher than observed for real-life cloud applications: We chose a relatively high crash probability to have a high enough likelihood to observe a few crashes during T, so we can study their effects on QoS measures.

$$CP_c = 0.5\% \quad \forall \; c \in C \tag{2.8}$$

2.3.5. Methodological Principles of Reproducibility

We followed the principles of reproducibility introduced in [68]:

- *Repeated experiments:* We repeated our scientific experiment with several runs, as explained in each chapter.
- *Workload and configuration coverage:* We covered many experimental cases and different configurations.
- *Experimental setup description:* We reported the technical details of our experiment. Moreover, in each chapter, we report our setup specific to the experiment performed.
- *Open-access artifact:* To support replicability, we published the data, code, and evaluation logs of our experiments as an open-access artifact in the online appendix of this dissertation[6].

- *Probabilistic result description of measured performance:* We described our empirical results in each chapter separately.
- *Statistical evaluation:* In each chapter, we performed a separate statistical evaluation of the results.
- *Measurement units:* We reported all units.
- *Cost:* We did not have any costs for the experiments as we performed them on our available private cloud. For the GCP validation experiments, we used the free tier services[11].

[11] https://cloud.google.com/free

3. Reliability Model

This chapter presents our analytical models of reliability to address the research problem P_1: *Lack of reliability models specific to service- and cloud-based dynamic routing.* We model component crashes and calculate the expected request loss by these crashes. To validate our models, we analyze the data from our scientific experiment. Our proposed reliability model is also used in Chapter 4 to study the specific trade-offs analysis of reliability and performance. Moreover, this chapter addresses P_3: *Lack of an approach to automatically adapt the reliability and performance trade-offs* as the presented model is used for the quality-of-service trade-offs adaptation in the later chapters of this thesis.

3.1. Introduction

Various dynamic routing architectures are used in today's service- and cloud-based architectures, including sidecar-based routing [49, 79], routing through a central entity such as an API gateway [79], or architectures with multiple dynamic routers [45]. So far, the impacts of such architectures and their different configurations on system reliability have not been extensively studied. Therefore, it is hard to consider reliability as a trade-off in the architectural design decision for more centralized or distributed dynamic request routing. While more centralized approaches tend to be easier to manage, understand and change, it is more difficult to reach a fine-grained level of control [61]. As reliability is a core consideration in service and cloud architectures [67], a reasonably accurate failure prediction for architecture design options is beneficial. This prediction would help architects better design system architectures considering quality-of-service trade-offs.

We model request loss during router and service crashes in an analytical model based on Bernoulli processes [90]. Request loss is the externally visible metric indicating the severity of the crashes' impacts. The model abstracts central entities, dynamic routers, and sidecars in a common router abstraction. This abstraction makes it possible to predict request loss during router and service crashes for any configuration of a request flow sequence in service- and cloud-based system architectures.

To validate our analytical model, we performed an extensive experiment of 1200 hours of runtime. We study 36 representative experiment cases (i.e., different experiment configurations) for the three kinds of architectures with different numbers of cloud services, routers, and request call frequencies (see Section 2.3 for more details). We compute the prediction error of our model compared to our empirical results. Our results show that the error is constantly reduced with more experiment runs, converging at a prediction error of 8.1%. Given the common target prediction accuracy of up to 30% in the cloud performance domain [59] and the fact that the goal of our study is architecting with a rough prediction

of impact on system reliability, these results are more than reasonable. With the same crash probability for all components, the same frequency of incoming requests, and the same number of cloud components, our model predicts, and our experiment confirms, that more decentralized routing results in losing a higher number of requests compared to more centralized approaches.

The structure of the chapter is as follows: Section 3.2 presents a specific metamodel and our analytical reliability model. Next, in Section 3.3, we describe the empirical validation

Table 3.1.: The Mathematical Notations Used in this Chapter

Notation	Description
T	Observed system time
n_{rout}	Number of routers
n_{serv}	Number of services
n_{crash}	Number of crash tests
CI	Crash interval
cf	Incoming call frequency
A	Allocation of routers
Com	Set of all components
$r_{crashed}$	A router r when crashed
$s_{crashed}$	A service s when crashed
IR	Internal request
IR_T	Total number of internal requests per a call sequence
IL_T	Total internal loss
IL_R	Sum of the internal loss per crash of each router
IL_S	Sum of the internal loss per crash of each service
IL_c	Internal loss for a component c
EL_T	Total external loss
C_T	Total number of crashes
n_c^{exec}	Number of executed internal requests for the crash of a component c
d_c	Expected average downtime after a component c crashes
CP_c	Crash probability of a component c every CI
$E[C_c]$	Expected number of crashes of a component c during T
r/s	Requests per second
$MAPE$	Mean absolute percentage error
MAE	Mean absolute error
MSE	Mean squared error
MSE	Root mean squared error
$model_c$	Result of the model for the experiment case c
$empirical_c$	Measured empirical data for the experiment case c
$Cases$	Set of experiment cases
n_c	Length of $Cases$

of our study. We evaluate the prediction error and discuss the threats to validity in Section 3.4. Finally, Section 3.5 concludes the chapter.

3.2. Model of Request Loss During Router and Service Crashes

In this section, firstly, we explain the central concepts of our work with a metamodel. Secondly, we propose a Bernoulli model [90] of request loss during router and service crashes. Table 3.1 presents the mathematical notations used in this chapter.

3.2.1. Metamodel

We consider various kinds of *Components* in service-based architectures as shown in Figure 2.4. In this chapter, we consider the following components: *API Gateways*, *Clients*, *Services*, and *Routers*. *Request* models the request flow, linking a source and a destination component. *Client Request* is an abstraction of a request flow between a *Client* and

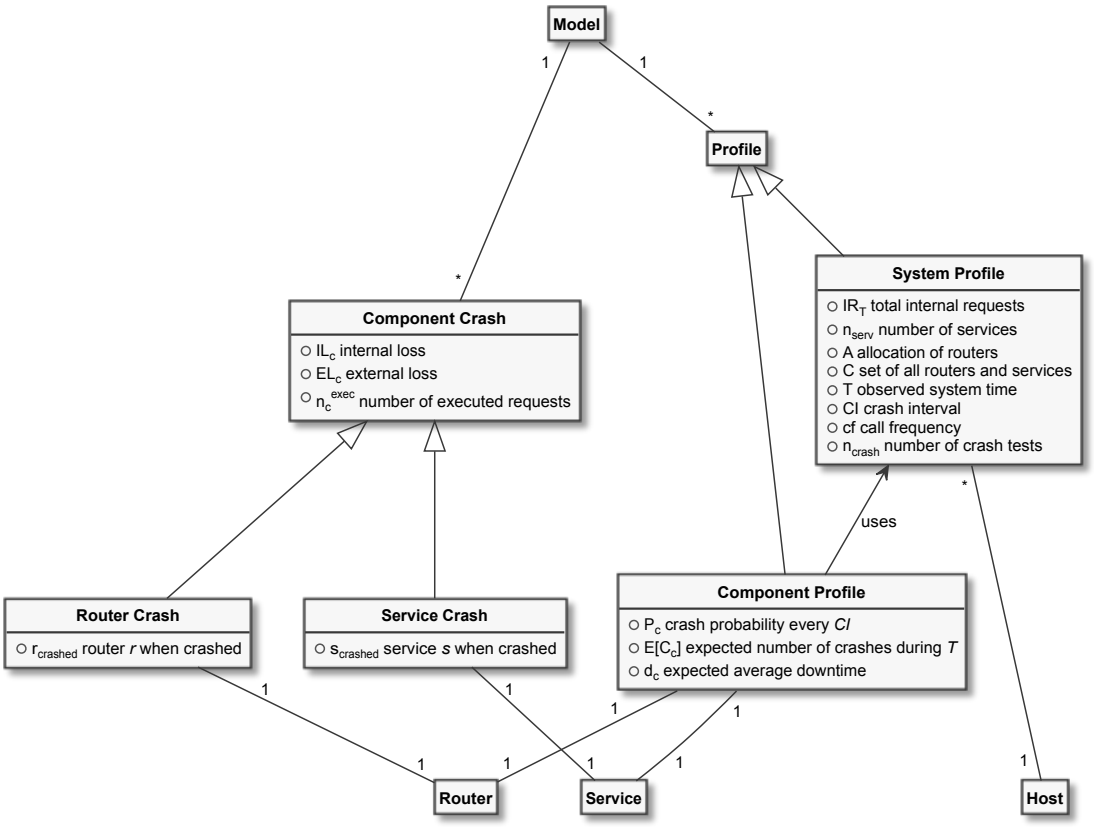


Figure 3.1.: Specific Metamodel Concepts for Modeling Request Loss

an *API Gateway*. *Internal Request (IR)* models a request flow amongst *API Gateway*, *Router*, and *Service* components. Figure 3.1 presents our extended metamodel with specific concepts for modeling request loss. The *Profile* and *Crash* classes contain member variables explained below in our model.

3.2.2. Definition of Internal and External Loss

To illustrate our approach, we use the concepts of our metamodel in Figure 2.4 to instantiate an example model. Figure 3.2 shows a configuration of the Adaptive Dynamic Routers (ADR) architecture with three routers and five services. The instantiated components send internal requests, labeled from IR1 to IR11, amongst one another to complete the processing of one client request. The partially ordered set representing the call trace, i.e., the Client Request, IR1, ..., IR11, is called the *call sequence*. When a router or a service crashes before it has processed a pending request, client requests will not be processed fully, which results in the application not being responsive to the client. We define *external loss* as the number of client requests not processed during a component crash and *internal loss* as the number of lost internal requests.

Internal Loss

Let IR_T be the total number of internal requests per call sequence. When a component crashes, the internal loss can be counted as IR_T minus the number of already-executed

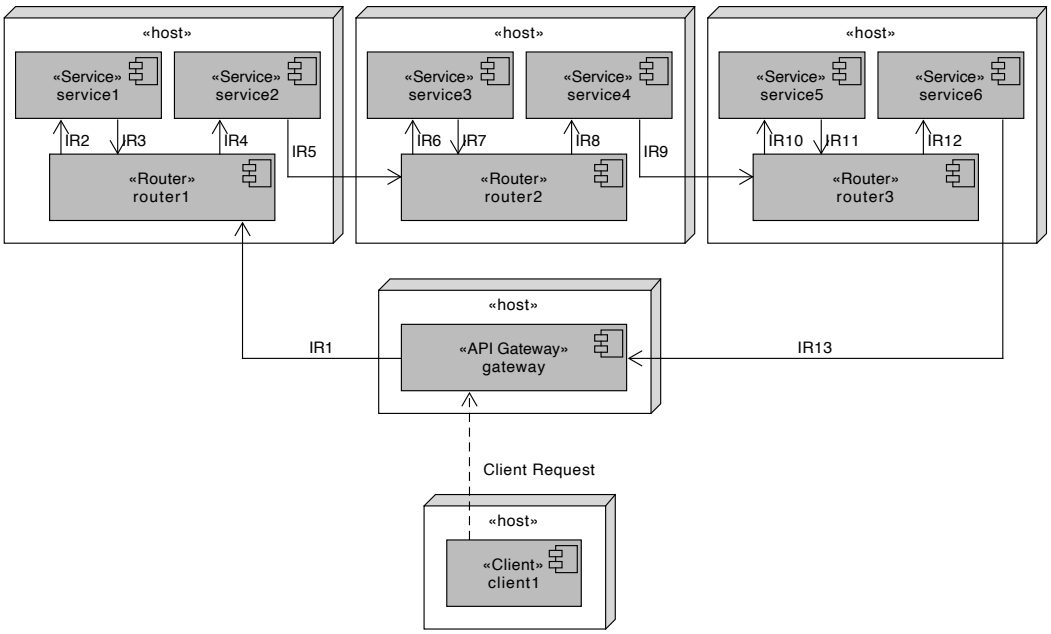


Figure 3.2.: Example Model Instance with Internal Requests

requests (per each external loss). Let IL_c, EL_c and n_c^{exec} be the internal loss, the external loss, and the number of executed internal requests for the crash of a component c:

$$IL_c = EL_c \cdot (IR_T - n_c^{exec}) \tag{3.1}$$

An Example Crash Scenario

To clarify, we consider the crash of *router*3 in Figure 3.2. In this case, IR1 to IR8 will be executed, i.e., eight executed internal requests. However, we lose five internal requests, namely IR9 to IR13. We can see that there are a total of thirteen internal requests:

$$n_c^{exec} = 8 \tag{3.2}$$
$$IR_T = 13 \tag{3.3}$$
$$IL_c = EL_c \cdot 5 \tag{3.4}$$

that means per each external loss, we lose five internal requests. Note that IR_T and n_c^{exec} must be parameterized based on the application. An example of this parameterization is given in Section 3.3.

External Loss

Let d_c be the expected average downtime after a component c crashes and cf the incoming call frequency, i.e., the frequency at which client requests are received. The following formula gives the external loss per crash of each component c. The external loss is calculated as the number of client requests that were not processed:

$$EL_c = d_c \cdot cf \tag{3.5}$$

3.2.3. Bernoulli Process to Model Request Loss During Router and Service Crashes

In this section, we model request loss based on Bernoulli processes [90]. Note that we only model the crash of routers and services sub-components in our metamodel. This is because we assume an API gateway is stable and reliable. Moreover, a crash of a client results in requests not being generated, so requests are not lost. Hence, we use the common term *components* for all instantiated routers and services throughout the rest of this chapter.

Number of Crash Tests

Let T be the observed system time. During T, all components can crash with certain failure distributions. It is realistic to assume that these distributions are known with a certain error, as they can be estimated from past system runs, e.g., recorded in system logs. Note that many cloud systems run without stopping. T should be interpreted as the interval during which these failure distributions are observed (e.g., failure distributions of

a day or a week). A crash of each component can happen at any time in T. We model this behavior by checking for a crash of any system components in intervals. Let CI be the crash interval, i.e., the period of time we check for a crash. Our model "knows" about crashes in discrete time intervals only, as would be the case, e.g., if the Heartbeat pattern [46] or the health check API pattern [76] are used for checking system health. Our model allows any possible values for T or CI and different crash probabilities for each component, e.g., based on empirical observations in a system under consideration. Let n_{crash} be the number of times we check for a crash of components during T, i.e., the number of crash tests:

$$n_{crash} = \lfloor \frac{T}{CI} \rfloor \tag{3.6}$$

Expected Number of Crashes

Each crash test is a Bernoulli trial in which success is defined as "component crashed" and failure as "component did not crash". Assuming $CI > d_c$, all n_{crash} crash tests of a *component* c are independent of each other. This assumption is justifiable since, in reality, when a component crashes and is down, it cannot crash again. Another crash of the same component can happen only after the component is up and running, i.e., the component's downtime has passed. Therefore, for each component, we can create a Bernoulli process of its crash tests. Then, the binomial distribution of each Bernoulli process gives us the number of successes [90], that is, the number of times a component crashes during T. For each component, the expected value of the binomial distribution is the expected number of crashes. Let CP_c be the crash probability of a component c every time we check for a crash which is derived, dependent on the application, from the failure distributions. Also, let $E[C_c]$ be the expected number of crashes of a component c during T:

$$E[C_c] = n_{crash} \cdot CP_c \tag{3.7}$$

Total Number of Crashes

Let C_T be the total number of crashes. C_T can be calculated as the sum of the expected number of crashes of each component.

$$C_T = \sum_{c \in Com} E[C_c] \tag{3.8}$$

which we can rewrite based on Equations (3.6) and (3.7) as:

$$C_T = \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in Com} CP_c \tag{3.9}$$

Total External Loss

Let EL_T be the total external loss. EL_T can be calculated as the sum of external loss per the crash of each system component:

$$EL_T = \sum_{c \in Com} E[C_c] \cdot EL_c \tag{3.10}$$

The total external loss can be rewritten using Equations (3.5) to (3.7) as:

$$EL_T = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} CP_c \cdot d_c \tag{3.11}$$

Total Internal Loss

Let IL_T be the total internal loss. IL_T can be calculated as the sum of internal loss per each component crash. Let Com be the set of all components that can crash, i.e., routers and services, we have:

$$IL_T = \sum_{c \in Com} E[C_c] \cdot IL_c \tag{3.12}$$

which can be rewritten using Equations (3.1) and (3.5) to (3.7) as:

$$IL_T = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} CP_c \cdot d_c \cdot (IR_T - n_c^{exec}) \tag{3.13}$$

3.3. Empirical Validation

This section introduces the experiment we designed to validate the accuracy of our reliability model. Moreover, we provide application-specific model formulae regarding our experiment setup. Finally, we present our empirical results.

3.3.1. Experiment Planning

We present the specific details of our experiment in this chapter. Section 2.3 reports the general details of our scientific experiment, e.g., the architecture configurations, the experiment cases, and the methodological reproducibility principles.

Goals

Our experiment aims to empirically validate our model's accuracy with regard to the number of crashes as well as the total external and internal loss represented by Equations (3.10) and (3.12). We realized the architectures using a prototypical implementation, instantiated and ran them in our private cloud infrastructure, measured the empirical results, and compared them with our model.

Specific Model Formulae

We parameterize different model elements based on our experiment cases. These parameterized models are used in this chapter to compare to the empirical measurements. Moreover, we use these models in the later chapters, where we compare different quality-of-service measurements. As explained before, in Equation (3.1), IR_T and n_c^{exec} need to be parameterized based on the application. In these configurations, each service receives an internal request, processes it, and sends it back to a router or the API gateway. We

can calculate IR_T based on the number of services as the following since there are one incoming and one outgoing request per each service and one request from the gateway:

$$IR_T = 2n_{serv} + 1 \tag{3.14}$$

In the example configuration presented in Figure 3.2, we have $n_{serv} = 6$ services and $IR_T = 13$ internal requests.

To calculate n_c^{exec}, we need to differentiate between service and router crashes, that is why we have different concepts for them in the metamodel shown in Figure 3.1. In case of a *service crash*, all internal requests until the last router will be executed. Let $s_{crashed}$ be the label number of the crashed service, then for our architecture configurations, we have:

$$n_c^{exec} = 2s_{crashed} - 1 \tag{3.15}$$

Remember that we use a uniform crash probability and downtime of components, as well as constant system time and crash interval (see Section 2.3.4). Using Equation (3.13), the internal loss for all services, i.e., IL_S, can be calculated as:

$$IL_S = 0.6 \cdot cf \cdot n_{serv}(n_{serv} + 1) \tag{3.16}$$

Let A be the allocation of routers, which is a set indicating the number of directly linked services of each router. In case of a *router crash*, we must know A. For instance, the allocation of routers in the example model presented in Figure 3.2 is:

$$A = \{2, 2, 2\} \quad and \quad A_0 = 0 \tag{3.17}$$

Let n_{rout} be the number of routers in a system. In our experiment, services were equally allocated to routers:

$$A = \{\frac{n_{serv}}{n_{rout}}, \frac{n_{serv}}{n_{rout}}, ..., (\frac{n_{serv}}{n_{rout}} \pm 1)\} \quad and \quad A_0 = 0 \tag{3.18}$$

in which A has the length of n_{rout}. In the example model instance, there are six services, i.e., $n_{serv} = 6$, and three routers, i.e., $n_{rout} = 3$. Therefore, we have the allocation presented in Equation (3.17).

Let $r_{crashed}$ be the label number of the crashed router. For our architecture configurations, we have:

$$n_c^{exec} = 2 \sum_{r=1}^{r_{crashed}} A_{r-1} \tag{3.19}$$

which means to find the number of executed requests before the crash of router r, we sum over the allocated services of all routers up until the crashed router and multiply it by two since there is one incoming and one outgoing request from a service to a router (see Figure 3.2). Let IL_R be the internal loss for all routers:

$$IL_R = 0.6 \cdot cf \cdot [n_{serv} + n_{rout}(n_{serv} + 1)] \tag{3.20}$$

Finally, we can rewrite Equation (3.12) by adding Equations (3.16) and (3.20) as:

$$IL_T = 0.6 \cdot cf \cdot [(n_{serv})^2 + (n_{rout} + 2)n_{serv} + n_{rout}] \tag{3.21}$$

We rewrite Equation (3.21) for the investigated architectures separately. In the case of the Central Entity (CE) architecture in our experiment, all services are connected to one router, i.e., the central entity component (see Section 2.1 for architecture details).

$$n_{rout} = 1 \tag{3.22}$$
$$IL_T = 0.6 \cdot cf \cdot [(n_{serv})^2 + 3n_{serv} + 1] \tag{3.23}$$

In the case of the Dynamic Routers (DR) architecture in our experiment, all n_{serv} services are equally distributed (with a maximum difference of one service) on the three dynamic routers:

$$n_{rout} = 3 \tag{3.24}$$
$$IL_T = 0.6 \cdot cf \cdot [(n_{serv})^2 + 5n_{serv} + 3] \tag{3.25}$$

In the case of the Sidecar-based Architecture (SA) in our experiment, each service is connected to one router, i.e., a sidecar.

$$n_{rout} = n_{serv} \tag{3.26}$$
$$IL_T = 0.6 \cdot cf \cdot [2(n_{serv})^2 + 3n_{serv}] \tag{3.27}$$

Since we use a constant system time and crash interval and a uniform crash probability and downtime of components, we can rewrite Equations (3.9) and (3.11) for our experiment using our cases reported in Section 2.3.4:

$$EL_T = 0.6 \cdot cf \cdot (n_{serv} + n_{rout}) \tag{3.28}$$
$$C_T = 0.2 \cdot (n_{serv} + n_{rout}) \tag{3.29}$$

Data Set Preparation

For each experiment case, we instantiated the architectures. We ran the experiment for exactly ten minutes (excluding setup time), during which we checked for crashes and logged the output so we could later calculate the number of external losses precisely. As outlined above, we studied three architectures, three levels of n_{serv}, and four levels of cf, resulting in 36 experiment cases. Therefore, a single run of our experiment takes exactly six hours (36×10 minutes) of runtime. Since our model revolves around expected values in Bernoulli processes, we repeated this process for 200 experiment runs, i.e., 1200 hours of runtime, and reported the arithmetic mean of the results. To support reproducibility, the code and evaluation scripts are provided in the online artifact of this dissertation[6].

3.3.2. Empirical Results

In this section, we present the predicted results of our analytical model and the empirical results shown in Table 3.2 and Figure 3.3.

IL_T is a model element that incorporates crashes of all components. Moreover, it includes all model views, e.g., architecture configurations, expected average downtime, etc. Therefore, we conduct our analysis mainly based on IL_T. It can be observed from Table 3.2 that when we keep n_{serv} constant, increasing cf results in a rise of EL_T in all cases, which leads to a higher value of IL_T.

Table 3.2.: Results of the Model and the Experiment

Arch.	n_{serv}	cf	C_T	EL_T	IL_T	C_T	EL_T	IL_T	$\sigma(IL_T)$
			Model			Experiment			
CE	**3**	**10**	0.800	24.000	114.000	0.760	23.395	98.960	118.552
		25	0.800	60.000	285.000	0.620	47.435	228.975	292.389
		50	0.800	120.000	570.000	0.705	106.370	480.235	608.635
		100	0.800	240.000	1140.000	0.725	218.130	1045.000	1216.765
	5	**10**	1.200	36.000	246.000	1.165	36.405	236.575	236.536
		25	1.200	90.000	615.000	1.110	85.400	608.040	574.267
		50	1.200	180.000	1230.000	1.115	172.085	1155.550	1173.295
		100	1.200	360.000	2460.000	1.040	317.585	2223.655	2101.272
	10	**10**	2.200	66.000	786.000	1.920	62.000	720.190	616.778
		25	2.200	165.000	1965.000	2.125	171.290	2063.305	1711.931
		50	2.200	330.000	3930.000	2.160	344.765	4223.665	3458.119
		100	2.200	660.000	7860.000	1.960	590.665	6853.500	6567.047
DR	**3**	**10**	1.200	36.000	162.000	1.075	32.505	153.045	175.952
		25	1.200	90.000	405.000	1.225	92.745	452.160	466.814
		50	1.200	180.000	810.000	1.225	182.595	882.695	916.540
		100	1.200	360.000	1620.000	1.130	328.925	1477.405	1470.332
	5	**10**	1.600	48.000	306.000	1.670	51.995	319.210	301.989
		25	1.600	120.000	765.000	1.760	135.105	816.895	686.709
		50	1.600	240.000	1530.000	1.790	270.540	1597.535	1324.199
		100	1.600	480.000	3060.000	1.635	490.990	2909.115	2353.168
	10	**10**	2.600	78.000	930.000	2.525	82.255	921.610	495.543
		25	2.600	195.000	2325.000	2.355	187.715	2181.590	1275.035
		50	2.600	390.000	4650.000	2.205	345.350	4043.070	2508.002
		100	2.600	780.000	9300.000	2.375	741.870	8544.700	5022.780
SA	**3**	**10**	1.200	36.000	162.000	1.140	34.910	170.265	186.911
		25	1.200	90.000	405.000	1.230	93.265	435.685	452.190
		50	1.200	180.000	810.000	1.215	181.305	883.510	911.088
		100	1.200	360.000	1620.000	1.185	345.950	1634.850	1844.829
	5	**10**	2.000	60.000	390.000	1.795	55.745	350.055	244.898
		25	2.000	150.000	975.000	1.795	138.910	891.525	647.402
		50	2.000	300.000	1950.000	1.715	261.740	1716.095	1284.733
		100	2.000	600.000	3900.000	1.790	528.420	3385.240	2633.592
	10	**10**	4.000	120.000	1380.000	3.900	127.715	1443.040	773.632
		25	4.000	300.000	3450.000	3.745	306.745	3477.305	1979.270
		50	4.000	600.000	6900.000	3.860	617.375	7140.655	4262.114
		100	4.000	1200.000	13800.000	3.870	1232.770	14072.910	8287.361

Since in our experiment, we instantiated the DR architecture with three dynamic routers, it is interesting to consider the experiment case of $n_{serv} = 3$. In this case, SA and DR have the same number of components, i.e., routers and services. Note that SA uses a sidecar per each cloud service; therefore, with $n_{serv} = 3$, we will also have three sidecars. The difference between the two architectures in this experiment case is that in DR, dynamic routers are placed on a different VM than their directly-linked services. However, in SA, sidecars are placed on the same VM on which their corresponding cloud services reside. For this reason, it can be observed that the reported values for SA and DR closely resemble each other when we have different values of cf but keep the number of cloud services n_{serv} constant at three. This resemblance can also be observed in Figures 3.3a, 3.3d, 3.3g and 3.3j, especially with higher values of cf, the distributions of IL_T and the interquartile ranges of DR and SA are very close to one another.

Considering the cases with five or ten cloud services, we almost always observe higher IL_T when we change the architecture from a CE to a DR or from a DR to an SA but keep the same configurations, that is, if we keep n_{serv} and cf constant. It is because, in our experiment, CE has only one control logic component (the central entity), DR has three (dynamic routers), and SA has n_{serv} (sidecars). Consequently, the number of crashes corresponding to control logic components goes up from CE to DR and then to SA. This increases the total number of crashes C_T (predicted in our model by Equation (3.8)), which results in losing more requests.

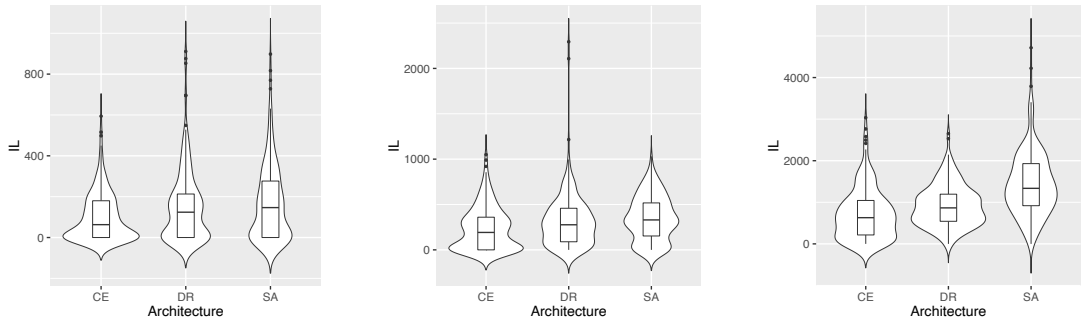
One interesting observation regarding these cases is that when $n_{serv} = 5$, we have one router for CE, three for DR, and five for SA. In Figures 3.3b, 3.3e, 3.3h and 3.3k, we can see in the violin plots and the boxplots that the difference in the internal loss between CE and DR is close to that of DR and SA. However, in the case of $n_{serv} = 10$, we observe a much higher internal loss for SA as shown in Figures 3.3c, 3.3f, 3.3i and 3.3l. This is because, as before, we have one router for CE and three for DR, but now ten for the SA architecture.

3.4. Discussion

In this section, we evaluate the error of the prediction. Moreover, we discuss the threats to the validity of our study.

3.4.1. Evaluation of the Prediction Error

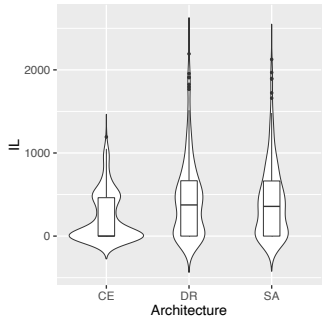
We use the predicted results of our model presented in Table 3.2 to measure the model's accuracy compared to the empirical data from our experiment. To do so, we measure the prediction error by calculating the Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE) [90]. Let $model_c$ and $empirical_c$ be the result of the model and the measured empirical data for the experiment case c, respectively. Also, let $Cases$ be the experiment cases and n_c the length of $Cases$, which is 36 in this study. We use the following formulae



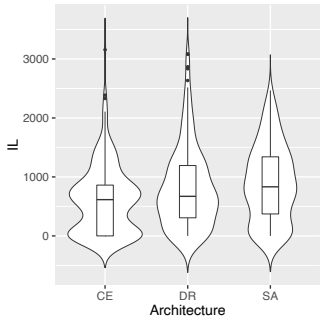
(a) $(n_{serv}, cf) = (3, 10)$

(b) $(n_{serv}, cf) = (5, 10)$

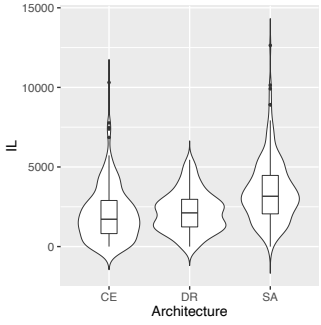
(c) $(n_{serv}, cf) = (10, 10)$



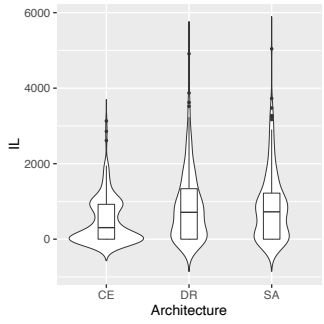
(d) $(n_{serv}, cf) = (3, 25)$



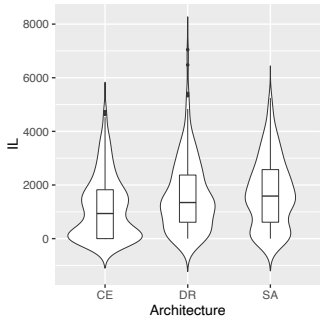
(e) $(n_{serv}, cf) = (5, 25)$



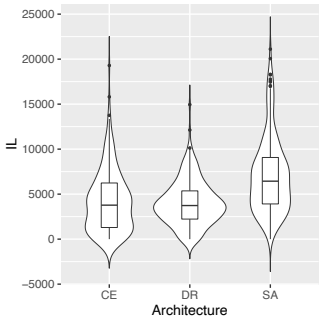
(f) $(n_{serv}, cf) = (10, 25)$



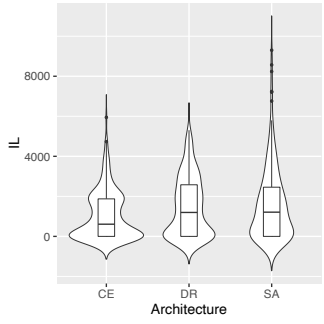
(g) $(n_{serv}, cf) = (3, 50)$



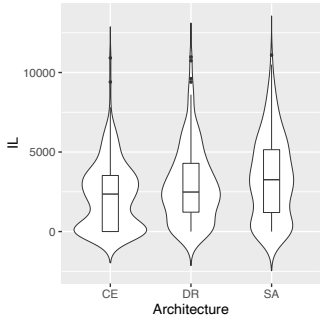
(h) $(n_{serv}, cf) = (5, 50)$



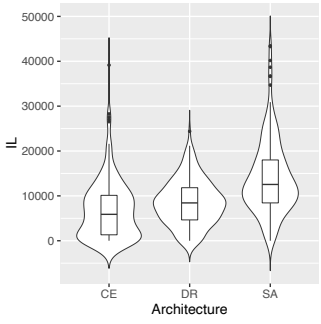
(i) $(n_{serv}, cf) = (10, 50)$



(j) $(n_{serv}, cf) = (3, 100)$



(k) $(n_{serv}, cf) = (5, 100)$



(l) $(n_{serv}, cf) = (10, 100)$

Figure 3.3.: Plots of all Experiment Cases Regarding the Total Internal Loss

for our error measurements:

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \left| \frac{model_c - empirical_c}{empirical_c} \right| \tag{3.30}$$

$$MAE = \frac{1}{n_c} \cdot \sum_{c \in Cases} |model_c - empirical_c| \tag{3.31}$$

$$MSE = \frac{1}{n_c} \cdot \sum_{c \in Cases} (model_c - empirical_c)^2 \tag{3.32}$$

$$RMSE = \sqrt{MSE} \tag{3.33}$$

By definition, the expected value is the mean of a large number of repetitions [39]. As previously mentioned, a single run of our experiment takes six hours of runtime (plus more than three hours of experiment setup and post-processing of the results). In total, we were able to run the experiment 200 times. Table 3.3 reports prediction error measurements of our model for a different number of runs regarding multiple model elements. A low number of repeats is expected to increase the error since the effects of outliers on the arithmetic mean of the data are considerable.

As the table shows, with a higher number of experiment runs, the prediction error is reduced, which indicates a converging error rate. After 200 runs, the final prediction MAPE error of 8.1% regarding IL_T is already low enough for the goal of this chapter, which is the use of our model for predictions during architectural decision-making, where even rough estimates would be beneficial. As mentioned, the common target prediction accuracy in the cloud performance domain is 30% [59].

3.4.2. Threats to Validity

As in all empirical research, there are several threats to the validity and limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [96].

3.4.3. Construct validity

In our study, we injected crashes to simulate real-world crash behavior at a given probability. While this is a common approach (see Section 1.3 for state of the art), the threat remains that measuring internal and external loss based on these crashes might not measure reliability well. For example, system reliability is also influenced by cascading effects of crashes beyond a single call sequence [67] that are not covered in our experiment. More research, probably with real-world systems and crashes, would be needed to exclude this threat.

3.4.4. Internal validity

We collected an extensive amount of data to validate our model. However, we did so in limited experiment time and with injected crashes, simulated by stopping Docker containers. We avoided factors such as other loads on the machines where the experiment

Table 3.3.: Prediction Error Measurements for Different Number of Experiment Runs

Error	Number of Runs	CT	ELT	ILT
MAPE (%)	**50**	12.919	12.307	13.946
	100	9.416	8.492	9.593
	150	8.326	7.426	8.731
	200	8.081	7.097	8.105
MAE	**50**	0.202	24.803	251.464
	100	0.154	18.471	205.340
	150	0.139	16.290	172.579
	200	0.130	16.743	158.877
MSE	**50**	0.057	1209.921	170988.061
	100	0.037	831.525	156649.362
	150	0.030	671.540	118297.046
	200	0.026	611.528	74579.457
RMSE	**50**	0.240	34.784	413.507
	100	0.192	28.836	395.790
	150	0.174	25.914	343.943
	200	0.160	24.729	273.092

ran, and much of the related literature takes a similar approach. Still, research observing real-world cloud-based systems with crashes would be needed to confirm that there are no other factors influencing the measurements.

3.4.5. External validity

To increase internal validity, we decided not to run the experiment on a public cloud where, e.g., other loads on the experiment machines might have had a significant impact on the results. Consequently, there is the threat that generalization to a public cloud setting might be limited. As our private cloud setting uses very similar hardware and software stacks as many public cloud offerings, we believe this threat to be small. The results might not be generalizable beyond the given experiment cases of 10-100 requests per second and call sequences of length 3-10. But as this covers a wide variety of loads and call sequences in cloud-based applications, the impact of this threat should be limited. A related threat is that we implemented our model instances with Node.js, not using off-the-shelf implementations, e.g., Envoy[1]. We did so to have a comparable infrastructure and to avoid technological impacts on our results.

[1] https://www.envoyproxy.io/

3.4.6. Conclusion Validity

As the statistical method to compare our model's predictions to the empirical data, we used the MAPE metric as it is widely used and offers good interpretability in our research context. To mitigate the threat that this statistical method might have issues, we double-checked three other error measures, i.e., MAE, MSE, and RMSE, which led to similar converging results.

3.5. Conclusions

In this chapter, we studied the impact of an architectural design decision regarding dynamic routing on system reliability and how this impact can be predicted. In particular, the important cases of central entity, dynamic routers, and sidecar-based architectures were investigated (see Section 2.1 for details). Our study concludes that more decentralized routing results in losing a higher number of requests than more centralized approaches. We derived an analytical model for predicting request loss in the studied architectures and empirically validated this model using 36 representative experiment cases. Our results indicate that, with more experiment runs, the prediction error is constantly reduced, converging at a prediction error of 8.1%. The major impact of our work is on architectural design decisions for the dynamic-routing architectures in service- and cloud-based systems. Before our work, architects had to rely solely on their experiences as empirical evidence focusing specifically on cloud-based dynamic routing was not available. To the best of our knowledge, our work is the first to provide such evidence.

Our analytical reliability model in this chapter is a central point in our study and is used repeatedly in the remainder of this dissertation. In the next chapter, we provide a detailed trade-offs analysis of reliability and performance. Moreover, we suggest an analytical performance model in the presence of component crashes (impeded reliability). Having defined this model, we provide an approach in Chapter 5 to automatically adapt the reliability and performance trade-offs. Furthermore, in Chapter 8, we study a multifaceted reconfiguration of dynamic routing applications, where we use our reliability model, introduced and empirically validated in this chapter.

4. Performance Models

This chapter presents performance models to address the research problem P_2: *Lack of performance models specific to service- and cloud-based dynamic routing.* Firstly, we perform multiple regression analyses on the round-trip times of the requests recorded during our experiment. We use this statistical performance model to precisely analyze the reliability and performance trade-offs of our private cloud infrastructure. Secondly, we present an analytical performance model that can be generalized to other infrastructures. This versatile analytical model is used in Chapter 5 for the quality-of-service trade-offs adaptation addressing P_3: *Lack of an approach to automatically adapt the reliability and performance trade-offs.*

4.1. Introduction

System reliability and performance are crucial quality attributes in almost all service- and cloud-based systems. Choosing the wrong architecture pattern or configuration may severely impact the reliability or performance of a software system. We aim to provide models and empirical evidence to precisely estimate the reliability and performance trade-offs. In the last chapter, we proposed an analytical model of request loss for reliability modeling. We studied the accuracy of this model's predictions empirically and calculated the error rate in 200 experiment runs, during which we recorded the round-trip time of requests. In this chapter, we use these values to create statistical performance models based on multiple regression analyses [80]. Our regression analyses of the performance data results in prediction models with high statistical significance. The results show that distributed approaches for dynamic data routing have a better performance compared to centralized solutions. We use the statistical performance models to precisely study the reliability and performance trade-offs on our private infrastructure. However, these statistical performance models do not apply to other infrastructures. Therefore, we propose an analytical performance model generalizable to cloud-based dynamic routing applications. We use our analytical model in the rest of this dissertation as it is more versatile. Our results provide important new insights into dynamic routing architecture decisions.

The structure of the chapter is as follows. Section 4.2 presents our statistical performance model. Section 4.3 gives a detailed reliability and performance trade-offs analysis. Section 4.4 presents our analytical model of performance, and Section 4.5 explains the empirical validation of analytical model. Section 4.6 discussed the threats to the validity of our study, and Section 4.7 concludes the chapter.

4.2. Statistical Model of Performance

This section presents performance models from the data of our experiment. Table 4.1 presents the mathematical notations used in this chapter. To support reproducibility, the code, data, and log of our study are presented in the online artifact of this dissertation[6].

Table 4.1.: The Mathematical Notations Used in this Chapter

Notation	Description
RTT	Round-Trip Time
SC	Service coefficient
FC	Frequency coefficient
IC	Interaction coefficient
Int	Intercept
$LinearReg.$	Linear regression analysis
$NonlinearReg.$	Nonlinear regression analysis
Q_1	First quartile
Q_3	Third quartile
$\sigma(RTT)$	Standard deviation of RTT
$\sigma(P)$	Standard deviation of performance
P	Performance model
P_{arch}	Performance model for each architecture
P_{CE}	Performance model for the central entity architecture
P_{DR}	Performance model for the dynamic routers architecture
P_{CE}	Performance model for the sidecar architecture
R	Reliability model
R_{arch}	Reliability model for each architecture
R_{CE}	Reliability model for the central entity architecture
R_{DR}	Reliability model for the dynamic routers architecture
R_{CE}	Reliability model for the sidecar architecture
T	Observed system time
n_{rout}	Number of routers
n_{serv}	Number of services
CI	Crash interval
cf	Incoming call frequency
Com	Set of all components
d_c	Expected average downtime after a component c crashes
CP_c	Crash probability of a component c every CI
Req	Number of client requests
r/s	Requests per second
$MAPE$	Mean absolute percentage error
MAE	Mean absolute error
MSE	Mean squared error
MSE	Root mean squared error
$model_c$	Result of the model for the experiment case c
$empirical_c$	Measured empirical data for the experiment case c
$Cases$	Set of experiment cases
n_c	Length of $Cases$

4.2.1. The Round-Trip Time

To compare and measure the performance of the architectures, we recorded the round-trip time of requests in our experiment. Let *RTT* be the round-trip time, which is defined as the difference in time from the moment the API gateway receives a request until it is routed through all cloud services involved in the processing of the request. Firstly, we generate an identification (ID) number for each HTTP request. Whenever the API gateway receives a request, it starts a timer with an attached ID. Next, the request is routed through cloud services and returns to the gateway. Finally, the gateway reads the request ID and stops the corresponding timer. The RTT is the time calculated by the timer.

4.2.2. Statistical Methods

Multiple regression analysis is a technique used to create prediction models that estimate the value of a *dependent variable* based on the values of two or more *independent variables* [80]. We consider the round-trip time of requests as the dependent variable and the number of services and call frequencies as the independent variables. Moreover, we create separate analyses for each architecture indicated by the number of routers. These dynamic-routing architectures include the central entity (CE), dynamic routers (DR), and sidecar-based architecture (SA) as presented in Section 2.1. The following hypotheses were formulated for this experiment:

H_0: There is no significant prediction accuracy of the round-trip time of requests by the number of services and call frequencies.

H_A: There is a significant prediction accuracy of the round-trip time of requests by the number of services and call frequencies.

We created two prediction models, i.e., linear and nonlinear, per each architecture configuration to estimate the *RTT* based on call frequency and the number of services.

Table 4.2.: Prediction Models of Performance

Arch.	Service Coefficient (SC)	Frequency Coefficient (FC)	Interaction Coefficient (IC)	Intercept (Int)	F-statistic: p-value
CE	3.384e+00	-3.042e-01	5.528e-02	1.608e+01	<2.2e-16
	7.343e+00	0.0265e+00	-	-7.599e+00	<2.2e-16
DR	4.881e+00	-1.254e-01	-1.509e-05	1.287e+01	<2.2e-16
	4.870e+00	-0.125e+00	-	12.872e+00	<2.2e-16
SA	3.360e+00	-0.034e+00	-0.011e+00	5.708e+00	<2.2e-16
	2.552e+00	-0.102e+00	-	10.540e+00	<2.2e-16

We used the R language[1] for our statistical analysis.

4.2.3. Prediction Models

Let n_{serv} be the number of services and cf the incoming call frequency of a dynamic-routing application. Also, let SC, FC, IC, and Int be the service coefficient, frequency coefficient, interaction coefficient, and intercept. Table 4.2 presents our prediction models for each architecture Our models result in a very low *p-value* (high statistical significance of the predicted results) which allows us to reject the null hypothesis and accept the alternative hypothesis indicating that the number of services and the call frequency affect the RTT.

$$LinearReg. = SC \cdot n_{serv} \quad + \quad FC \cdot cf \quad + \quad Int \tag{4.1}$$

$$NonlinearReg. = SC \cdot n_{serv} \quad + \quad FC \cdot cf \quad + IC \cdot n_{serv} \cdot cf \quad + \quad Int \tag{4.2}$$

The interaction term in Equation (4.2), i.e., $IC \cdot n_{serv} \cdot cf$, tells us that the effect of the number of services on the predicted RTT is not constant. It changes with different values of call frequency (and vice versa). Note that regression models are calculated from all 200 runs of our experiment.

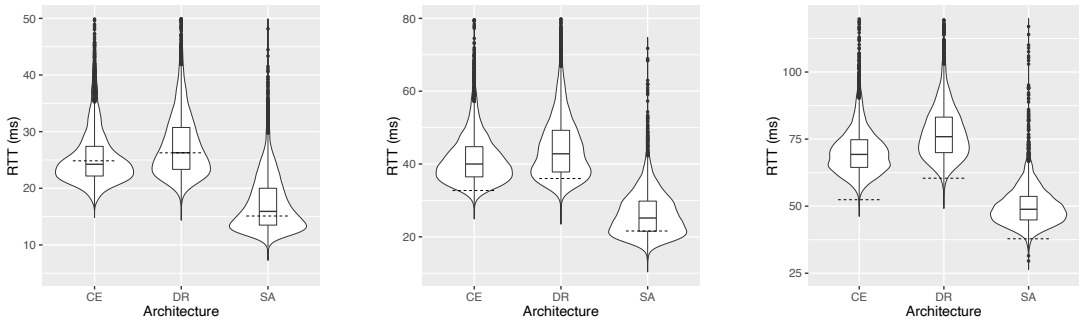
4.2.4. Empirical Results

Table 4.3 and fig. 4.1 compares the empirical data with the predicted results. Let Q_1 and Q_3 be the first and third quartiles of the recorded round-trip times, and $\sigma(RTT)$ the standard deviation of the empirical data. We report the first quartile, the median, the third quartile, the 95th percentile, the mean, and the standard deviation of RTT. We can observe that the predictions in the case of DR and SA lie within the interquartile range of the empirical data in most cases. Exceptions are the following cases with a call frequency of 10 r/s: DR with five and ten services and SA with $n_{serv} = 10$. In these cases, the nonlinear prediction is slightly below the first quartile of the empirical data. Moreover, the predicted RTTs in the case of DR with $n_{serv} = 10$ and $cf = 50$ r/s is above Q_3. With CE, the nonlinear predicted results are closer to the arithmetic mean of the data than to the median, as also confirmed in Table 4.4 with the lower prediction error of 13.7% compared to 19.3%. Note that the common target prediction accuracy in the cloud performance domain is 30% [59].

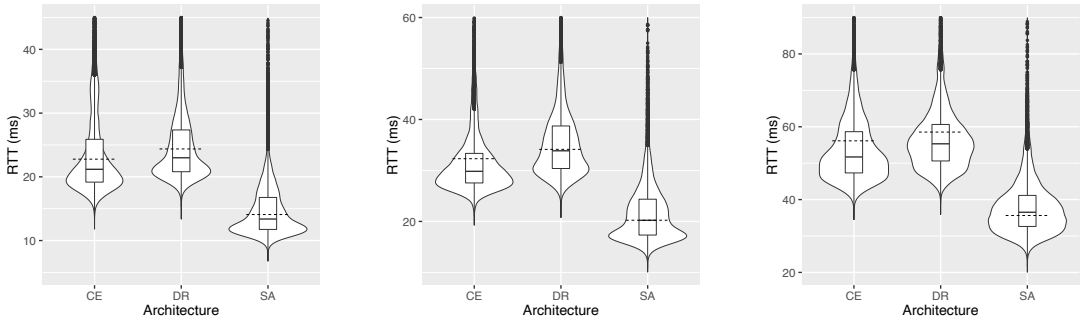
4.2.5. Evaluation of the Prediction Error

We compare the results of our prediction models to another run of our experiment (not used in the training set). Table 4.4 presents the prediction error of the regression models. The nonlinear regression compared to the arithmetic mean of the empirical data results in a lower prediction error. We use the Mean Absolute Percentage Error (MAPE) [90]. Let $model_c$ and $empirical_c$ be the result of the model, the measured empirical data for

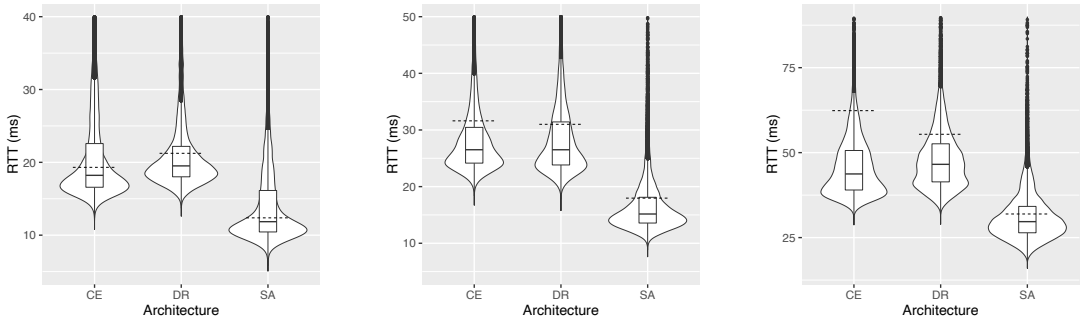
[1] https://www.r-project.org



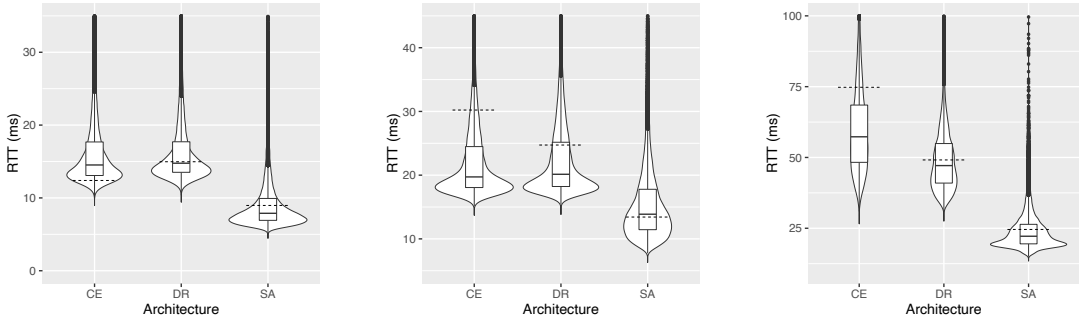
(a) $(n_{serv}, cf) = (3, 10)$ (b) $(n_{serv}, cf) = (5, 10)$ (c) $(n_{serv}, cf) = (10, 10)$



(d) $(n_{serv}, cf) = (3, 25)$ (e) $(n_{serv}, cf) = (5, 25)$ (f) $(n_{serv}, cf) = (10, 25)$



(g) $(n_{serv}, cf) = (3, 50)$ (h) $(n_{serv}, cf) = (5, 50)$ (i) $(n_{serv}, cf) = (10, 50)$



(j) $(n_{serv}, cf) = (3, 100)$ (k) $(n_{serv}, cf) = (5, 100)$ (l) $(n_{serv}, cf) = (10, 100)$

Figure 4.1.: The RTT and the Nonlinear Regressions (Dashed Lines) for all Cases

Table 4.3.: Comparison of the Prediction Results of the Performance Models and the Empirical Data

Arch.	n_{serv}	cf (r/s)	Q_1 (ms)	Median RTT (ms)	Q_3 (ms)	95th Percentile (ms)	Mean RTT (ms)	$\sigma(RTT)$	Linear Regression (ms)	Nonlinear Regression (ms)
CE	3	10	22.173	24.277	27.504	36.627	26.0169	11.899	14.695	24.848
		25	19.228	21.327	26.773	39.951	24.423	9.466	15.093	22.773
		50	16.618	18.339	23.367	35.350	21.333	9.863	15.756	19.314
		100	13.101	14.597	17.983	27.975	16.938	9.843	17.083	12.396
	5	10	36.490	40.021	44.845	55.381	42.138	15.835	29.381	32.722
		25	27.564	29.862	33.428	44.942	31.987	12.791	29.780	32.305
		50	24.185	26.618	30.752	42.250	28.948	11.276	30.442	31.610
		100	18.078	19.794	24.810	35.657	23.966	24.713	31.769	30.220
	10	10	64.488	69.357	74.901	88.528	72.344	30.946	66.096	52.406
		25	47.363	51.796	58.966	72.632	55.832	33.015	66.494	56.135
		50	39.035	43.826	50.811	63.718	48.306	37.599	67.158	62.350
		100	48.634	58.066	70.423	95.812	74.398	139.257	68.484	74.780
DR	3	10	23.371	26.374	30.955	40.017	28.322	11.521	26.257	26.259
		25	20.845	23.152	27.744	38.264	25.477	9.504	24.374	24.377
		50	18.053	19.601	22.588	35.026	21.901	9.295	21.237	21.241
		100	13.536	14.817	18.005	28.168	17.192	10.349	14.962	14.968
	5	10	37.844	42.893	49.4277	62.270	45.422	18.780	36.016	36.020
		25	30.442	34.011	39.034	51.303	36.345	14.731	34.133	34.138
		50	23.863	26.637	31.799	43.272	29.350	15.122	30.996	31.001
		100	18.242	20.235	25.503	36.201	23.584	16.343	24.721	24.727
	10	10	70.034	76.020	83.473	97.357	79.636	36.074	60.414	60.424
		25	50.677	55.427	60.877	75.861	58.545	29.661	58.532	58.541
		50	41.436	46.638	52.788	65.423	51.010	47.884	55.394	55.402
		100	40.997	47.254	55.167	70.112	54.562	75.960	49.119	49.125
SA	3	10	13.500	15.938	20.042	26.399	17.427	6.483	17.176	15.106
		25	11.747	13.381	16.782	22.975	14.881	5.155	15.648	14.083
		50	10.449	11.875	16.258	25.607	14.188	6.349	13.102	12.377
		100	6.923	7.898	9.975	18.061	9.456	5.196	8.010	8.965
	5	10	21.554	25.185	29.860	37.137	26.561	10.007	22.279	21.601
		25	17.330	20.227	24.383	33.295	21.881	7.671	20.751	20.239
		50	13.573	15.174	18.158	27.103	16.831	6.913	18.205	17.968
		100	11.456	13.896	17.857	27.665	15.726	7.908	13.113	13.427
	10	10	44.875	48.860	53.678	63.075	50.705	18.464	35.037	37.838
		25	32.633	36.5545	41.214	53.287	38.577	16.120	33.509	35.628
		50	26.433	29.718	34.265	45.468	32.117	18.422	30.963	31.946
		100	19.509	22.221	26.482	37.321	25.646	27.174	25.871	24.582

a case c, $Cases$ be the set of experiment cases, and n_c the length of $Cases$ (36 in our experiment).

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \left| \frac{model_c - empirical_c}{empirical_c} \right| \tag{4.3}$$

Table 4.4.: Prediction Error of the Performance Models

Regression	Empirical Data	CE (%)	DR (%)	SA (%)
Linear	Mean	21.527	8.966	10.343
	Median	25.483	9.902	11.119
Nonlinear	Mean	13.654	8.959	10.158
	Median	19.270	9.915	8.958

4.3. Reliability and Performance Trade-Off Analysis

So far, we described a reliability model in Chapter 3 and presented statistical performance models of Adaptive Dynamic Routers (ADR) applications. In this section, we analyze the trade-offs of the architectures concerning the two qualities in different combinations of configurations:

$$1 \leq n_{serv} \leq 10 \tag{4.4}$$
$$1 \leq cf \leq 100 \tag{4.5}$$

4.3.1. Reliability Comparison

We use the reliability models provided in Section 3.3 "Specific Model Formulae." Let R_{arch} be the analytical reliability model for each architecture:

$$R_{CE} = 0.6 \cdot cf \cdot [(n_{serv})^2 + 3n_{serv} + 1] \tag{4.6}$$
$$R_{DR} = 0.6 \cdot cf \cdot [(n_{serv})^2 + 5n_{serv} + 3] \tag{4.7}$$
$$R_{SA} = 0.6 \cdot cf \cdot [2(n_{serv})^2 + 3n_{serv}] \tag{4.8}$$

which are plotted in Figure 4.2. CE results in an equal or higher reliability than SA and DR. There are some cases where SA gives higher reliability than DR, especially in the lower ranges of n_{serv}. We study the architectures more precisely.

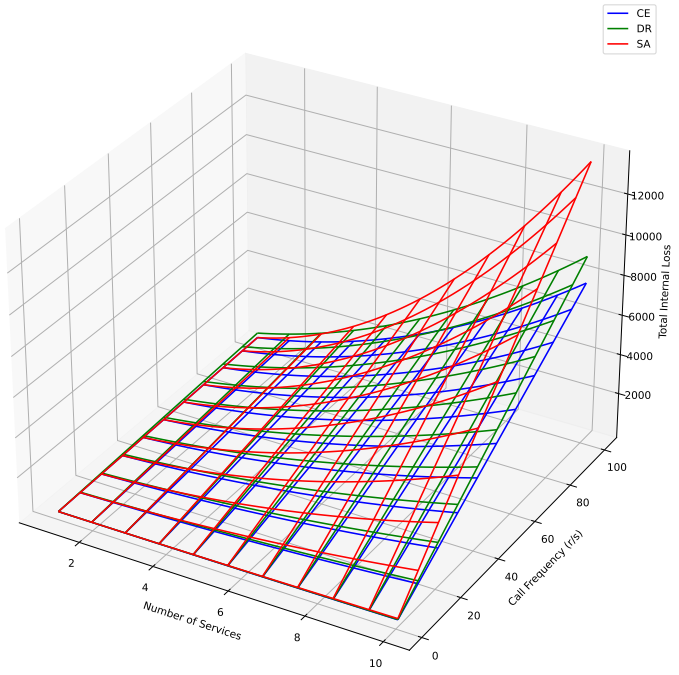


Figure 4.2.: Reliability Models

Reliability Trade-Off Between CE and DR

There is no combination of cf and n_{serv} where $R_{CE} = R_{DR}$. Therefore, CE always results in higher reliability than DR in our focused context.

Reliability Trade-Off Between CE and SA

We find the intersecting line where $P_{CE} = P_{SA}$ in our focused context is $n_{serv} = 1$. That is when we have only one service. Since we use the prototypical implementations for all architectures, SA and CE configurations become the same application. Therefore, they give the same reliability value. In any other case, CE results in higher reliability than SA.

Reliability Trade-Off Between DR and SA

We find the intersecting line where $P_{DR} = P_{SA}$ in our focused context is $n_{serv} = 3$. That is when there are three services, DR and SA are the same application in our implementation since they both have the same number of routers. Therefore, they result in the same reliability value. In our experiment, we instantiated DR with three and SA with n_{serv} routers. When $n_{serv} < 3$, SA has fewer routers than DR. Consequently, SA results in a lower number of request loss, i.e., higher reliability, than DR. When $n_{serv} > 3$, DR has fewer routers and results in higher reliability than SA.

Summary of the Reliability Trade-Offs

Table 4.5 summarizes the comparison of the architectures. A lower R_{arch} means fewer request losses. When $n_{serv} \leq 3$, the CE reliability predictions are lower or equal to those of SA, which are less or equal to the DR predictions for all call frequencies. When $n_{serv} > 3$, there is a different trend: The CE predictions are lower or equal to those of DR, which is less or equal to the SA predictions for all call frequencies.

Table 4.5.: Comparison of the Reliability of the Architectures

n_{serv}	cf (r/s)	Reliability
$n_{serv} \leq 3$	all	$R_{CE} \leq R_{SA} \leq R_{DR}$
$n_{serv} > 3$	all	$R_{CE} < R_{DR} < R_{SA}$

4.3.2. Performance Comparison

For the performance models, we used the nonlinear regression, i.e., Equation (4.2), in which the coefficients are taken from Table 4.2. Let P_{arch} be the performance prediction model for each architecture:

$$P_{CE} = 3.384 \cdot n_{serv} - 0.3042 \cdot cf + 16.08 + 0.05528 \cdot n_{serv} \cdot cf \tag{4.9}$$

$$P_{DR} = 4.881 \cdot n_{serv} - 0.1254 \cdot cf + 12.87 - 0.00001509 \cdot n_{serv} \cdot cf \tag{4.10}$$

$$P_{SA} = 3.360 \cdot n_{serv} - 0.0340 \cdot cf + 5.708 - 0.011 \cdot n_{serv} \cdot cf \tag{4.11}$$

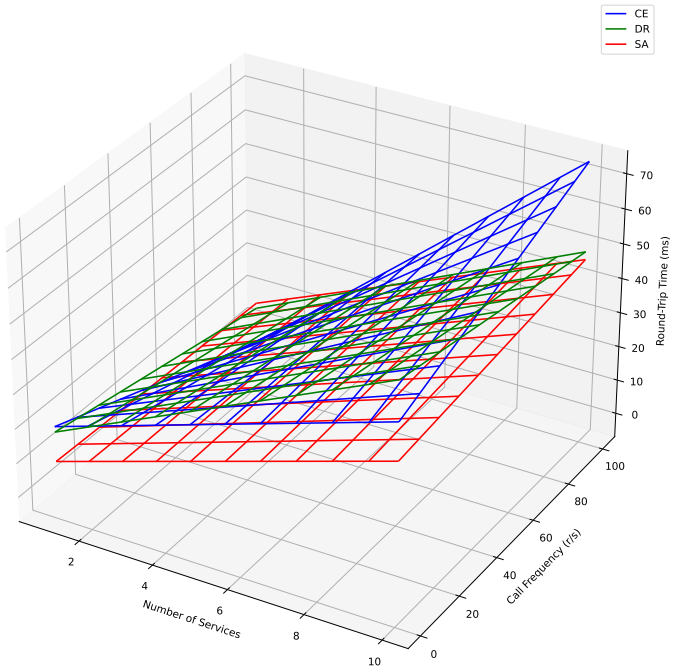
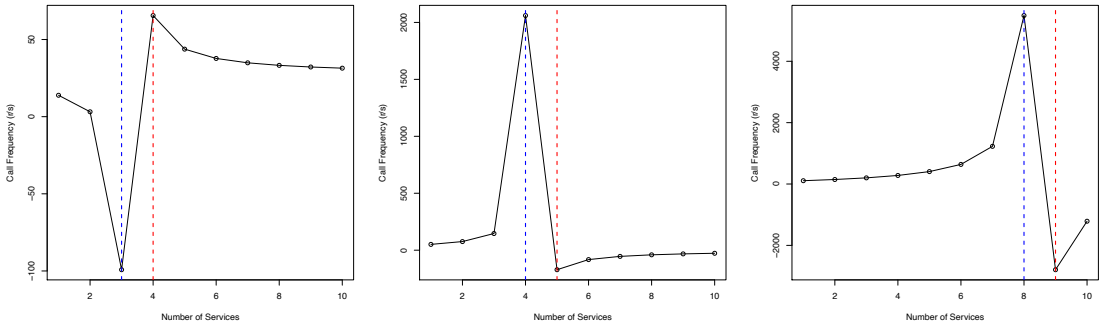


Figure 4.3.: Performance Models

The performance models are plotted in Figure 4.3. In most cases, SA results in a lower RTT than the other architectures. However, there are some cases that CE outperforms DR and SA. We compare the architectures to find the exact range of n_{serv} and cf, in which each architecture performs the highest.



(a) CE vs. DR (b) CE vs. SA (c) DR vs. SA

Figure 4.4.: Plot of All Intersecting Lines

Performance Trade-Off Between CE and DR

To characterize the trade-off more precisely, we have to study the intersecting line where $P_{CE} = P_{DR}$, i.e., the line where the curves of the architectures collide:

$$cf = \frac{1.497 \cdot n_{serv} - 3.21}{0.0552951 \cdot n_{serv} - 0.1788} \tag{4.12}$$

which is plotted in Figure 4.4a. Note that the blue dashed line, i.e., $n_{serv} = 3$, and the red dashed line, i.e., $n_{serv} = 4$, indicate the extrema of the intersecting line; therefore, CE outperforms DR in the area above the intersecting line when $n_{serv} \leq 3$, and below the intersecting line when $n_{serv} > 4$.

Table 4.6 summarizes the regions of cf and n_{serv}, in which CE outperforms DR. It can be confirmed by the results of our model for the experimental cases (see Table 4.3), in which under nonlinear regression, we can observe that in case of $n_{serv} = 3$, CE outperforms DR for all values of cf. However, when we have five or ten services, only in the lower range of incoming call frequency, i.e., 10 and 25, CE results in a lower performance value.

Table 4.6.: The Region Where CE outperforms DR

n_{serv}	1	2	3	4	5	6	7	8	9	10
cf (r/s)	≥ 13.87	≥ 3.17	≥ 1.00	≤ 65.55	≤ 43.77	≤ 37.73	≤ 34.90	≤ 33.26	≤ 32.19	≤ 31.43

Performance Trade-Off Between CE and SA

We find the intersecting line where $P_{CE} = P_{SA}$ in our focused context:

$$cf = \frac{-0.024 \cdot n_{serv} - 10.372}{0.06628 \cdot n_{serv} - 0.2702} \tag{4.13}$$

plotted in Figure 4.4b. In our focused context, CE outperforms SA only with the following conditions:

$$n_{serv} = 1 \quad and \quad cf \geq 50.98 \tag{4.14}$$

$$n_{serv} = 2 \quad and \quad cf \geq 75.71 \tag{4.15}$$

Performance Trade-Off Between DR and SA

The intersecting line where $P_{DR} = P_{SA}$ is plotted in Figure 4.4c:

$$cf = \frac{-1.521 \cdot n_{serv} - 7.162}{0.010985 \cdot n_{serv} - 0.091} \tag{4.16}$$

Summary of the Performance Trade-Offs

Table 4.7 gives a summary of performance analysis, in which lower P_{arch} means lower RTT, i.e., better performance.

Table 4.7.: Comparison of the Performance of the Architectures

n_{serv}	cf (r/s)	Performance
1	until 13.87	$P_{SA} \leq P_{DR} \leq P_{CE}$
	between 13.87 and 50.98	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 50.98	$P_{CE} \leq P_{SA} \leq P_{DR}$
2	until 3.17	$P_{SA} \leq P_{DR} \leq P_{CE}$
	between 3.17 and 75.71	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 75.71	$P_{CE} \leq P_{SA} \leq P_{DR}$
3	all	$P_{SA} \leq P_{CE} \leq P_{DR}$
4	until 65.55	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 65.55	$P_{SA} \leq P_{DR} \leq P_{CE}$
5	until 43.77	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 43.77	$P_{SA} \leq P_{DR} \leq P_{CE}$
6	until 37.73	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 37.73	$P_{SA} \leq P_{DR} \leq P_{CE}$
7	until 34.90	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 34.90	$P_{SA} \leq P_{DR} \leq P_{CE}$
8	until 33.26	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 33.26	$P_{SA} \leq P_{DR} \leq P_{CE}$
9	until 32.19	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 32.19	$P_{SA} \leq P_{DR} \leq P_{CE}$
10	until 31.43	$P_{SA} \leq P_{CE} \leq P_{DR}$
	from 31.43	$P_{SA} \leq P_{DR} \leq P_{CE}$

4.4. Analytical Performance Model

The statistical models presented in the last sections allowed us to precisely analyze the trade-offs on our private infrastructure. However, these models are not generalizable and applicable to other infrastructures such as Google Cloud Platform (GCP)[2]. In this section, we propose an analytical performance model of average request processing time per router. This model allows us to quantify the impact of dynamic-routing architecture patterns on performance in the presence of component crashes (impeded reliability). Our analytical performance model is general and not specific to our infrastructure. Therefore, it applies to public clouds as well.

4.4.1. Bernoulli Process to Model Request Loss During Crashes

In Chapter 3, we modeled the request loss based on Bernoulli processes [90]. Let R be reliability, T the observed system time, CI the crash interval, Com the set of components,

[2] https://cloud.google.com/

i.e., routers and services, CP_c the crash probability of a component c every CI, and d_c the expected average downtime of a component c after it crashes. Remember that cf is the incoming call frequency based on requests per second (r/s). We use the external loss function given by Equation (3.11):

$$R = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} CP_c \cdot d_c \tag{4.17}$$

In this formula, request loss is defined as the number of client requests not processed due to a failure, such as a component crash. Equation (4.17) gives the request loss as a reliability metric by calculating the expected value of the number of crashes. Having this information, we sum up all the lost requests during the downtime of a component.

4.4.2. Average Request Processing Time per Router

We model the average processing time of requests per router as a performance metric. This metric is important as it allows us to study the quality-of-service factors, such as the efficiency of architecture configurations. The total number of client requests, i.e., Req, is the call frequency cf multiplied by the observed time T:

$$Req = cf \cdot T \tag{4.18}$$

The number of processed requests is the total number of client requests minus the request loss. Let P be performance and n_{rout} the number of routers in an ADR application. The average processing time of requests per router is calculated so that we divide the total system time T over the processed requests and the number of routers:

$$P = \frac{T}{n_{rout} \cdot (Req - R)} \tag{4.19}$$

Using Equations (4.17) to (4.19), the average processing time is:

$$P = \frac{T}{n_{rout} \cdot cf \cdot \left(T - \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in Com} CP_c \cdot d_c\right)} \tag{4.20}$$

Note that our analytical model has some assumptions outlined in the threats to validity (see Section 4.6).

4.5. Empirical Validation of the Analytical Model

In this section, we introduce an experiment to empirically validate the accuracy of our performance model.

4.5.1. Experiment Planning

We present the specific details of our experiment in this chapter. Section 2.3 reports the general details of our scientific experiment, e.g., the architecture configurations, the experiment cases, and the methodological reproducibility principles.

Goals

We aim to empirically validate the accuracy of our analytical performance model represented by Equation (4.20). We realized the dynamic-routing architecture patterns (see Section 2.1 for details) using a prototypical implementation, ran them on private and public cloud infrastructures, measured the empirical results, and compared them with our model.

Validation Experiment on a Public Cloud

We used our private cloud to have control over the infrastructure. On a public cloud, other factors can influence the results, such as the parallel workload of other applications. To show that our approach can be used on other cloud platforms as well, we empirically validated the analysis of our proposed model on GCP. We used 7 instances of a compute-optimized C2 machine type[3], each with 4 vCPUs and 16 GB of memory. We duplicated our private cloud infrastructure on this machine and repeated the experiment for a public cloud validation run.

Data Set Preparation

During each experiment case, we logged the number of processed requests and calculated the average request processing time per router based on Equations (4.19) and (4.20). We studied four levels of cf, three levels of n_{serv}, and three levels of n_{rout}, resulting in 36 experiment cases. A single run of our experiment takes exactly 6 hours (36×10 minutes) of runtime. Since our model revolves around expected values in Bernoulli processes, we repeated this process for 200 experiment runs on our private cloud infrastructure, i.e., 1200 hours of runtime, and reported the arithmetic mean of the results. Moreover, for public cloud validation, we performed an experiment run of 6 hours on GCP. Overall, we had an extensive empirical validation of 1206 hours of runtime (excluding setup time).

Specific Model Formulae

As we had a uniform crash probability and downtime of each component in our experiment, we can rewrite the performance given by Equation (4.20) in ms:

$$P = \frac{1000}{n_{rout} \cdot cf \left(1 - (n_{rout} + n_{serv}) \cdot \lfloor \frac{T}{CI} \rfloor \cdot \frac{1}{T} \cdot CP_c \cdot d_c\right)} \tag{4.21}$$

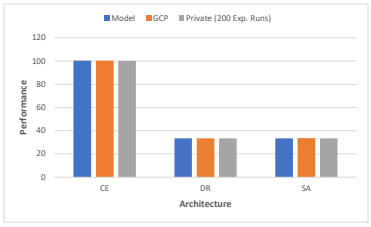
Using our experiment values reported in Section 2.3.4, we have:

$$P = \frac{1000}{n_{rout} \cdot cf(1 - 0.001(n_{serv} + n_{rout}))} \tag{4.22}$$

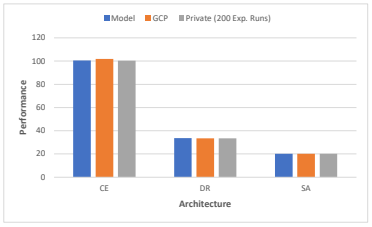
[3] https://cloud.google.com/compute/docs/compute-optimized-machines

Table 4.8.: Model Predictions and Empirical Measurements of Performance

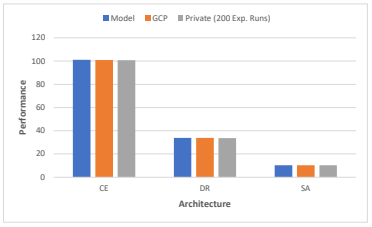
n_{rout} Arch. Pattern	n_{serv}	cf	P			$\sigma(P)$
			Model	GCP	Private (200 Exp. Runs)	
1 CE	3	10	100.40	100.33	100.13	0.37
		25	40.16	40.13	40.04	0.16
		50	20.08	20.15	20.01	0.08
		100	10.04	10.03	10.01	0.04
	5	10	100.60	101.83	100.33	0.54
		25	40.24	40.36	40.11	0.20
		50	20.12	20.14	20.06	0.11
		100	10.06	9.98	10.02	0.05
	10	10	101.11	101.01	100.80	0.86
		25	40.44	40.25	40.37	0.38
		50	20.22	20.18	20.19	0.18
		100	10.11	10.10	10.08	0.08
3 DR	3	10	33.53	33.43	33.43	0.19
		25	13.41	13.53	13.37	0.08
		50	6.71	6.65	6.68	0.04
		100	3.35	3.36	3.34	0.02
	5	10	33.60	33.43	33.53	0.2
		25	13.44	13.45	13.42	0.09
		50	6.72	6.68	6.71	0.04
		100	3.36	3.34	3.35	0.02
	10	10	33.77	33.80	33.68	0.22
		25	13.51	13.38	13.47	0.10
		50	6.75	6.72	6.72	0.04
		100	3.38	3.53	3.37	0.02
n_{serv} SA	3	10	33.53	33.58	33.42	0.19
		25	13.41	13.48	13.38	0.08
		50	6.71	6.87	6.68	0.04
		100	3.35	3.36	3.34	0.02
	5	10	20.20	20.16	20.13	0.12
		25	8.08	8.07	8.05	0.05
		50	4.04	4.01	4.02	0.02
		100	2.02	2.02	2.01	0.01
	10	10	10.20	10.31	10.21	0.19
		25	4.08	3.99	4.08	0.04
		50	2.04	2.03	2.04	0.02
		100	1.02	1.01	1.02	0.01



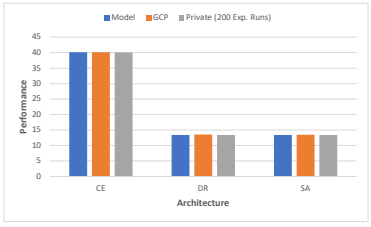
(a) $n_{serv} = 3$
$cf = 10$ r/s



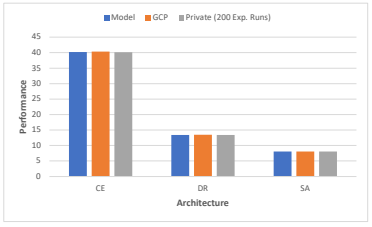
(b) $n_{serv} = 5$
$cf = 10$ r/s



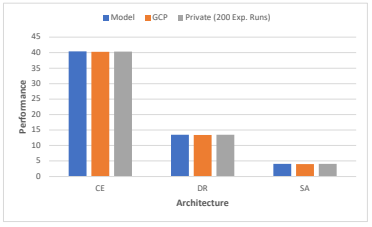
(c) $n_{serv} = 10$
$cf = 10$ r/s



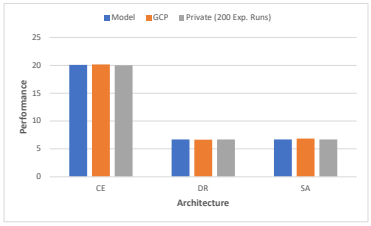
(d) $n_{serv} = 3$
$cf = 25$ r/s



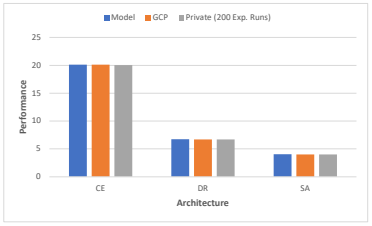
(e) $n_{serv} = 5$
$cf = 25$ r/s



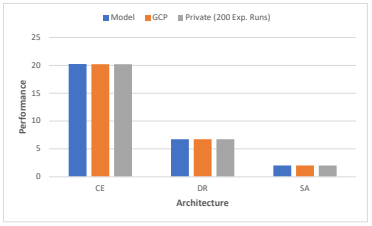
(f) $n_{serv} = 10$
$cf = 25$ r/s



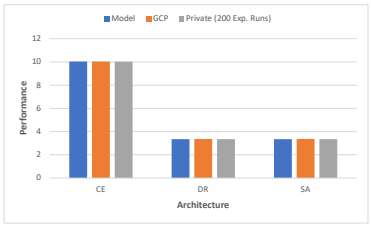
(g) $n_{serv} = 3$
$cf = 50$ r/s



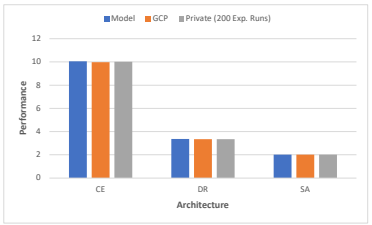
(h) $n_{serv} = 5$
$cf = 50$ r/s



(i) $n_{serv} = 10$
$cf = 50$ r/s



(j) $n_{serv} = 3$
$cf = 100$ r/s



(k) $n_{serv} = 5$
$cf = 100$ r/s

(l) $n_{serv} = 10$
$cf = 100$ r/s

Figure 4.5.: Plots of Model Predictions and Empirical Measurements

4.5.2. Experiment Results

Private Cloud Infrastructure We present the predicted results of our analytical model and the empirical measurements of our private infrastructure. In Table 4.8, we report the mean performance and its standard deviation $\sigma(P)$ of 200 experiment runs. As predicted by our model (see Equation (4.22)) and confirmed by our experiment data, the call frequency cf and the number of routers affect the performance conversely. When taking the same configuration, i.e., keeping n_{serv} and n_{rout} constant, increasing cf results in a lower average processing time per router in all cases. Moreover, the more routers in an ADR application, the lower the performance. This is expected as the performance is defined in our study as the average request processing time per router. The low standard deviations in all cases indicate that the performance data of 200 runs are clustered around the mean performance.

Validation Experiment on GCP

Our performance model predictions compared to the empirical measurements of the GCP public cloud and the private cloud infrastructure are shown in Figure 4.5. As can be seen, all of the empirical measurements are very close to the predictions. We further investigate the prediction error of our model.

4.5.3. Evaluation of the Prediction Error

We measure the accuracy of our model predictions. The prediction error is calculated using the four error measurements commonly used in the cloud quality-of-service research [90], i.e., Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE), Mean Squared Error (MSE) and Root Mean Squared Error (RMSE). The error measurements are calculated in terms of performance P. Remember that $model_c$ and $empirical_c$ are the result of the model and the measured empirical data for an experiment case c, $Cases$ is the set of experiment cases, and n_c is the length of $Cases$. In the error measurements, we average over the $n_c = 36$ experiment cases (see Section 4.5). We use the following formulae for our error measurements:

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \left| \frac{model_c - empirical_c}{empirical_c} \right| \tag{4.23}$$

$$MAE = \frac{1}{n_c} \cdot \sum_{c \in Cases} |model_c - empirical_c| \tag{4.24}$$

$$MSE = \frac{1}{n_c} \cdot \sum_{c \in Cases} (model_c - empirical_c)^2 \tag{4.25}$$

$$RMSE = \sqrt{MSE} \tag{4.26}$$

Table 4.9 presents the prediction error of our performance model (using the values reported in Table 4.8). Our model has a MAPE prediction error of 0.64% on GCP and 0.28 % on our private infrastructure. Averaged over public and private clouds, we have

Table 4.9.: Predictions Errors of the Performance Model

	GCP	**Private (200 Exp. Runs)**	**overall**
$MAPE\,(\%)$	0.64	0.28	0.46
$MASE$	0.10	0.06	0.08
MSE	0.05	0.01	0.03
$RMSE$	0.22	0.10	0.16

a very low error rate of 0.46%. Other low error measurements also confirm the high accuracy of our prediction model.

4.6. Threats to Validity

As in all empirical research, there are several threats to the validity and limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [96].

4.6.1. Construct Validity

We did not consider component overload that can influence the overall system's performance, e.g., when an overloaded component is non-responsive to incoming requests. As the self-adaptivity of service- and cloud-based systems is at a mature level in today's industrial tools, e.g., using Google Kubernetes[4] autoscaling, we believe this threat to be small. For the sake of simplicity, we made some assumptions when designing our analytical model of performance, which is common when modeling a real-world phenomenon. We did not consider the requests that were in the system already at the time of a crash of a component as we restarted all containers for each experiment run to increase internal validity (see the next section). We assumed that the crash probabilities are known based on the observed system logs in the past and checked for crashes. This is a common practice in real-world systems, e.g., when the Heartbeat pattern [46] or the Health Check API pattern [76] is used for checking system health. Moreover, we considered a generic downtime of the components and did not study metrics such as mean time to failure and recovery.

4.6.2. Internal Validity

Internal validity concerns factors that affect the independent variables with respect to causality. We collected an extensive amount of data to validate our model on public and private cloud infrastructures. However, we did so in a limited experiment time using simulated crashes by stopping Docker containers. More research observing real-world cloud-based systems for a longer period of time would be needed to confirm that there are no other factors influencing the measurements. One such factor is other workloads being

[4] https://kubernetes.io

processed simultaneously on the same infrastructure. We studied this factor by running a validation experiment on the Google Cloud Platform and showed that our model is applicable. As we used the standard technology stack offered by most cloud providers (see Section 2.3.2), we believe our results are representative of the service- and cloud-based applications.

4.6.3. External Validity

External validity concerns threats that limit the ability to generalize the results beyond the experiment. We designed our approach with generality in mind and explained how architects could specify our model to their needs (see Section 4.5). In spite of the fact that we evaluated our approach by designing a representative experiment and measuring empirical data, the threat remains that evaluating based on another infrastructure may lead to different results. To mitigate this thread, we validated our measurements on GCP infrastructure and showed that our results are applicable. Moreover, the results might not be generalizable beyond the given experimental cases of 10-100 requests per second and call sequences of length 3-10. As this covers a wide variety of loads and call sequences in cloud-based applications, the impact of this threat should be limited.

In our experiments, we considered a uniform crash probability for all components. This is a common assumption made in such experiments (see, e.g., [72, 73]) to increase the control over the experiment's dependent variables and, thus, the internal validity of the experiment. At the same time, this might decrease the external validity if the crash profiles observed in a real-world application are substantially different (see [84] for the trade-offs between the internal and external validity in empirical software engineering). To mitigate this threat, our model, in its general form, does not assume a uniform crash probability for all components. A related threat is that we implemented our model instances with Node.js, not using off-the-shelf implementations, e.g., Envoy[5]. We did so to have a comparable infrastructure and to avoid technological impacts on our results.

4.6.4. Conclusion Validity

Conclusion validity concerns factors that affect the ability to draw conclusions about the relations between treatments and study outcomes. As the statistical method to compare our model's predictions to the empirical data, we used the MAPE metric as it is widely used and offers good interpretability in our research context. To mitigate the threat that this statistical method might have issues, we double-checked three other error measures, i.e., MAE, MSE, and RMSE, that similarly confirmed the high accuracy of our prediction model.

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

4.7. Conclusions

In this chapter, we investigated three representative service and cloud architecture patterns for dynamic routing regarding their impact and trade-offs on reliability and performance. We studied what the performance impact is on service- and cloud-based dynamic routing, how we can predict these results, and if there is an optimal trade-off between architectures in terms of performance and reliability. We created prediction models that estimate the performance impact of the investigated architectures. The found models show high statistical significance. Moreover, we precisely calculated the range of the incoming call frequency and the number of services, where each architecture gives better results. Concerning system reliability, centralized routing results in a lower request loss. However, the sidecar-based architecture performs better, especially with more services. Dynamic routers can be seen as a middle ground, particularly with a higher number of services.

To make our performance model generalizable, we also proposed an analytical model of performance that considers the average processing time of requests per router in ms. We used Bernoulli processes [90] to predict the number of request losses during crashes. Having this information, we calculated the number of processed requests and divided the observed system time by this number. For the empirical validation of our model, we designed an extensive experiment of 1200 hours of runtime (excluding setup time). The prediction error indicates the very high accuracy of our performance model. To ensure that our model is generalizable, we ran a validation experiment of 6 hours on Google Cloud Platform[3] and showed that our predictions are applicable and generalizable (see Section 4.5.3). We double-checked the accuracy with three other error measurements that confirmed the results.

The major impact of our work is on architectural design decisions for dynamic routing. Our proposed analytical model is representative of service- and cloud-based systems and can be used in other environments and applications to give insight to architects. To the best of our knowledge, our work is the first to provide empirical evidence on the trade-off analysis of reliability and performance in cloud-based dynamic routing. Our work's main contributions are models and empirical research of widely used architectures. Such empirical studies enable the construction of new algorithms and architectures based on a solid and well-founded understanding of the existing architectures. To be successful, such works require careful empirical studies laying the foundation for understanding the existing state of the art and its limitations, providing ground truths, and offering data sets for further studies (such as our open access data set in the online artifact of this dissertation[6]).

5. Trade-Offs Adaptation

This chapter presents our approach to automatically adapt reliability and performance trade-offs to address the research problem P_3: *Lack of an approach to automatically adapt the reliability and performance trade-offs.* In this chapter, we use our reliability model presented in Chapter 3 and the analytical performance model presented in Chapter 4 to study the trade-off adaptations. This chapter also relates to P_6: *Lack of tool support for the multifaceted reconfiguration of dynamic routing applications.*

5.1. Introduction

Dynamic routing is an essential part of service- and cloud-based applications. Routing architectures are based on vastly different implementation concepts, such as API gateways [79], enterprise service buses [26], message brokers [45], or sidecars [49, 79]. However, their basic operation is that these technologies dynamically route or block incoming requests. This chapter proposes the details of our approach that abstracts all these routing patterns using our proposed Adaptive Dynamic Routers (ADR) architecture. We hypothesize that a self-adaptation of the dynamic routing is beneficial over any fixed architecture selections concerning reliability and performance trade-offs. Our approach dynamically self-adapts between more central or distributed routing to optimize system reliability and performance. This adaptation is calculated using a Multi-Criteria Optimization (MCO) analysis [4]. We evaluate our ADR architecture by analyzing our previously-measured data during an experiment of 1200 hours of runtime (see Section 2.3 for details). Our extensive systematic evaluation of 4356 cases confirms that our hypothesis holds and our approach is beneficial in terms of reliability and performance. Even on average, where right and wrong architecture choices are analyzed together, our novel architecture offers a 9.82% reliability gain and a 47.86% performance gain.

The structure of the chapter is as follows: Section 5.2 explains the details of our analysis, and Section 5.3 provides the tool that supports our concepts. Section 5.4 presents the evaluation of the presented approach, and Section 5.5 discusses the threats to the validity of our research. We conclude the chapter in Section 5.6.

5.2. Approach Details

This section presents our reconfiguration algorithm to adapt the reliability and performance trade-offs. Table 5.1 presents the mathematical notations used in this chapter.

Table 5.1.: The Mathematical Notations Used in this Chapter

Notation	Description
n_{rout}	Number of routers
n_{serv}	Number of services
T	Observed system time
r/s	Requests per second
cf	Incoming call frequency in r/s
EL_T	External request loss
R	Reliability model
$R_{n_{rout}}$	Reliability prediction of a respective architecture configurations
R_{th}	Reliability threshold
P	Performance model
$P_{n_{rout}}$	Performance prediction of a respective architecture configurations
P_{th}	Performance threshold
PW	Performance weight
$RGain$	Reliability gain in %
$PGain$	Performance gain in %
$Cases$	Set of experiment cases
n_c	Length of $Cases$

5.2.1. Reliability and Performance Models

In Chapters 3 and 4, we introduced analytical reliability and performance models. Let T be the observed system time, EL_T the external request loss of dynamic routing applications, i.e., the number of requests lost during T, given by Equation (3.28). To study the adaptation of reliability and performance trade-offs, we use the average request loss per second (r/s) as a reliability model. Let R be the reliability model, cf the incoming call frequency, n_{serv} the number of services, and n_{rout} the number of routers of an ADR application.

$$EL_T = 0.6 \cdot cf \cdot (n_{serv} + n_{rout}) \tag{5.1}$$

$$R = \frac{EL_T}{T} \tag{5.2}$$

$$R = 0.001 \cdot cf \cdot (n_{serv} + n_{rout}) \tag{5.3}$$

Moreover, we use the parameterized performance model in ms given by Equation (4.22). Let P be the performance model:

$$P = \frac{1000}{n_{rout} \cdot cf \cdot (1 - 0.001(n_{serv} + n_{rout}))} \tag{5.4}$$

5.2.2. Multi-Criteria Optimization Analysis

In our approach, the reconfiguration between the architecture configurations is performed based on an MCO analysis [4] automatically. Consider the following optimization problem: An application using the ADR architecture has n_{serv} services and is under stress for a period of time with the call frequency of cf. To optimize reliability and performance, the system can change between different architecture configurations dynamically by adjusting n_{rout}, ranging from a centralized routing ($n_{rout} = 1$) and up to the extreme of one router per service ($n_{rout} = n_{serv}$).

We use the notations $R_{n_{rout}}$ and $P_{n_{rout}}$ to specify the reliability and performance of the respective architecture configurations by their number of routers. For instance, only configuring one router R_1 indicates the reliability model of centralized routing, and configuring n_{serv} routers (i.e., $R_1, \ldots, R_{n_{serv}}$) indicates completely distributed routing. Let R_{th} and P_{th} be the reliability and performance thresholds, respectively. The MCO question is: Given a cf and n_{serv}, what is the optimal number of routers that minimizes request loss and average processing time of requests per router without the predicted model values violating their respective thresholds?

$$
\begin{aligned}
&Minimize \\
&\quad R_{n_{rout}} && (5.5)\\
&\quad P_{n_{rout}} && (5.6)\\
&Subject\ to \\
&\quad R_{n_{rout}} \leq R_{th} && (5.7)\\
&\quad P_{n_{rout}} \leq P_{th} && (5.8)\\
&\quad 1 \leq n_{rout} \leq n_{serv} && (5.9)
\end{aligned}
$$

Typically, there is no single answer to an MCO problem. Using the above analysis, we find a range of n_{rout} configurations, all meeting the constraints. One end of this range optimizes reliability and the other performance. Thus, we need a preference function so our approach can automatically select a final n_{rout} value.

5.2.3. Preference Function

An architect defines an importance vector that gives weights to reliability and performance. Using this vector, the preference function instructs the ADR architecture to choose a final n_{rout} value in the range found by the MCO analysis (see Algorithm 1). Let us consider an example: When performance is of the highest importance to an application, an architect gives the highest weight, i.e., 1.0, to performance and the lowest weight, i.e., 0.0, to reliability. Thus, the preference function chooses the highest value on the n_{rout} range to choose more distributed routing. This reconfiguration results in processing client requests in parallel, resulting in higher performance.

5.2.4. Automatic Reconfiguration

As explained in Chapter 2, the QoS monitor reads the monitoring data from the API gateway and feeds this information to the reconfiguration manager. This manager reconfigures the infrastructure or reschedules the containers. Algorithm 1 presents our reconfiguration algorithm. The QoS monitor triggers the algorithm, e.g., whenever reliability or performance metrics degrade. Time intervals, manual triggering or change in the incoming load can also be used to trigger the algorithm if more appropriate than metrics degradation (see Figure 2.6).

`reconfigureRouters(reconfigSolution)` in Algorithm 1 performs the final reconfiguration based on the chosen solution by either reconfiguring the infrastructure using the IaC component or rescheduling the containers using the container scheduler. Our supporting tool presented in Section 5.3 provides a simple implementation of this step.

5.2.5. Illustrative Sample Case

Let us consider an example application of centralized routing ($n_{rout} = 1$) with ten services ($n_{serv} = 10$). This sample case is operational for the expected call frequency of $cf = 100\ r/s$ with a reliability threshold of $R_{th} = 1.5\ r/s$ and a performance threshold of $P_{th} = 35\ ms$. Let PW be the performance weight. We study the case where performance

Algorithm 1: Reconfiguration Algorithm to Adapt the Reliability and Performance Trade-Offs

Input: R_{th}, P_{th}, performanceWeight

$R_{n_{rout}}, P_{n_{rout}}, cf, n_{serv} \leftarrow$ **consumeMonitoringData()**
routersRange $\leftarrow$ **MCO**($cf, n_{serv}, R_{n_{rout}}, P_{n_{rout}}, R_{th}, P_{th}$)
reconfigSolution $\leftarrow$ **preferenceFunction**(routersRange, performanceWeight)
reconfigureRouters(reconfigSolution)

```
function preferenceFunction(range, PW)
begin
    length ← max(range) - min(range) +1
    floor ← ⌊ PW * length ⌋
    if floor == max(range) then
        return max(range)
    else if floor == 0 then
        return min(range)
    else
        return floor + min(range) -1
    end
end
```

is of the highest importance, i.e., performance weight is given as $PW = 1.0$. We do the MCO analysis by rewriting Equations (5.3) and (5.4) for these values:

$$\begin{aligned} &Minimize \\ &\quad R_{n_{rout}} = 1 + 0.1 \cdot n_{rout} && (5.10) \\ &\quad P_{n_{rout}} = \frac{1000}{n_{rout} \cdot (99 - 0.1 \cdot n_{rout})} && (5.11) \\ &Subject\ to \\ &\quad R_{n_{rout}} \leq 1.5 && (5.12) \\ &\quad P_{n_{rout}} \leq 55\ ms && (5.13) \\ &\quad 1 \leq n_{rout} \leq 10 && (5.14) \end{aligned}$$

In Equation (5.10), the constraint on the reliability threshold of $R_{n_{rout}} \leq 1.5$ gives the highest value for the number of routers as $n_{rout} = 5$. Equation (5.11) informs that the performance predictions in the range of $1 \leq n_{rout} \leq 10$ always satisfy the performance threshold. Therefore, the acceptable range for n_{rout} is given as:

$$1 \leq n_{rout} \leq 5 \quad (5.15)$$

Since performance is of the highest importance ($PW = 1.0$), the preference function chooses the highest possible value of the number of routers in this range. Consequently, the final reconfiguration solution is $n_{rout} = 5$.

Remember that the initial configuration was the central routing, i.e., one router. In this case, we have the following reliability and performance predictions:

$$\begin{aligned} n_{rout} &= 1 && (5.16) \\ R_1 &= 1.1\ r/s && (5.17) \\ P_1 &= 10.11\ ms && (5.18) \end{aligned}$$

After the reconfiguration, we have five routers:

$$\begin{aligned} n_{rout} &= 5 && (5.19) \\ R_5 &= 1.5\ r/s && (5.20) \\ P_5 &= 2.03\ ms && (5.21) \end{aligned}$$

5.3. Tool Overview

We developed a prototypical tool to demonstrate our adaptive architecture. The tool is available in the online artifact of this dissertation[6].

5.3.1. Architecture

Figure 5.1 shows the tool architecture. We provide two modes, i.e., deployment and visualization. In the case of deployment, our tool generates artifacts in the form of Bash[1]

[1] https://www.gnu.org/software/bash/

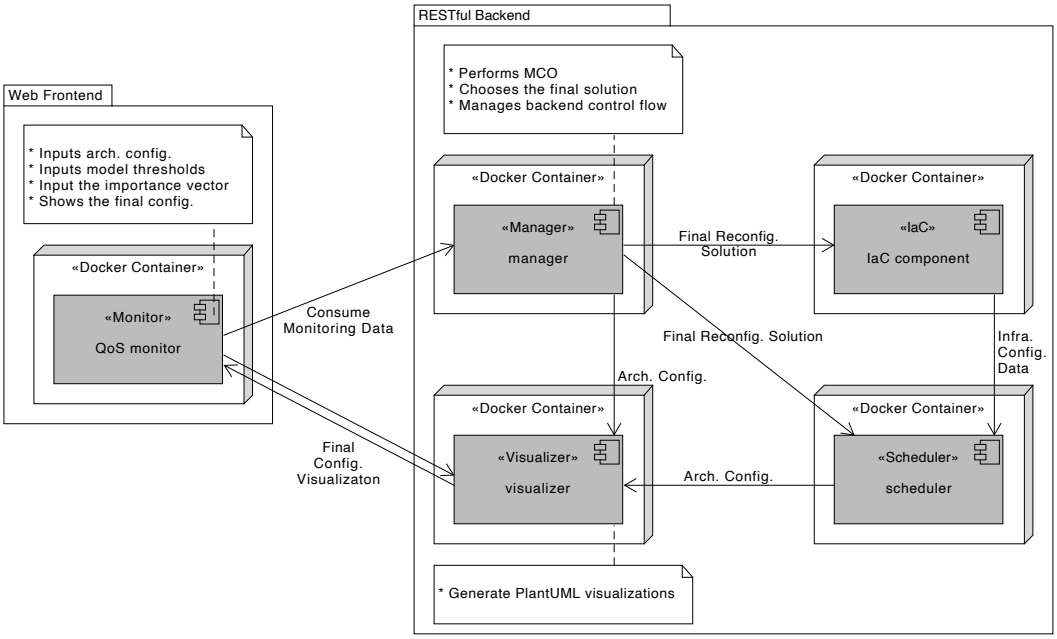


Figure 5.1.: Tool Architecture Diagram

scripts and configuration files, e.g., infrastructure configuration data to be used by an IaC tool. These scripts can be used to schedule containers using the Docker technology[2]. To study different scenarios, we also provide a visualization environment that only generates diagrams using PlantUML[3].

The frontend of our application provides the functionalities of the QoS monitor, i.e., to specify architecture configurations and model elements such as reliability and performance thresholds. This information is sent to the manager component in the backend that finds the final reconfiguration solution (see Algorithm 1). The manager sends this solution to the IaC component and the scheduler to generate deployment artifacts. A visualization is then created in the backend and shown in the frontend. The frontend is implemented in React[4] and the backend is developed in Node.js[5] as a RESTful application.

5.3.2. Toolflow

We divide the flow of our application into two parts, i.e., model creation and model reconfiguration. Figure 5.2 shows the flow for the model creation. An architect specifies the architecture configuration by entering the number of services and routers. Moreover, they choose the mode, i.e., deployment or visualization. In the case of deployment, the

[2] https://www.docker.com/
[3] https://plantuml.com/
[4] https://reactjs.org/
[5] https://nodejs.org/

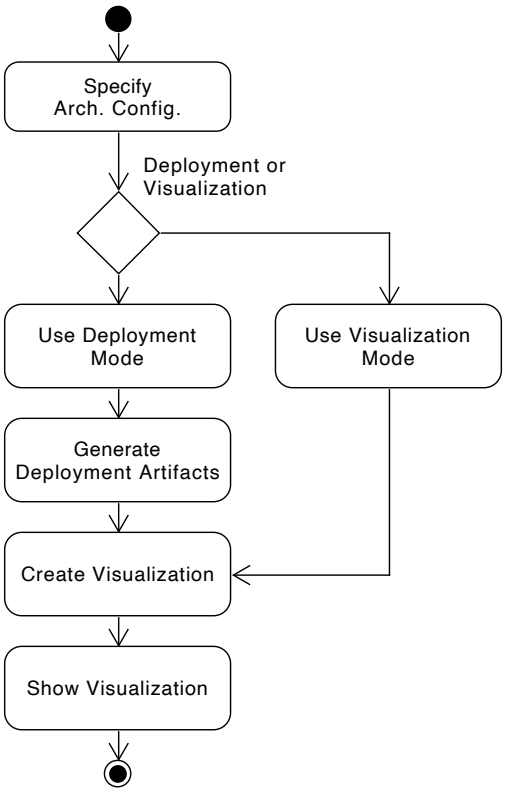


Figure 5.2.: Model Creation Toolflow

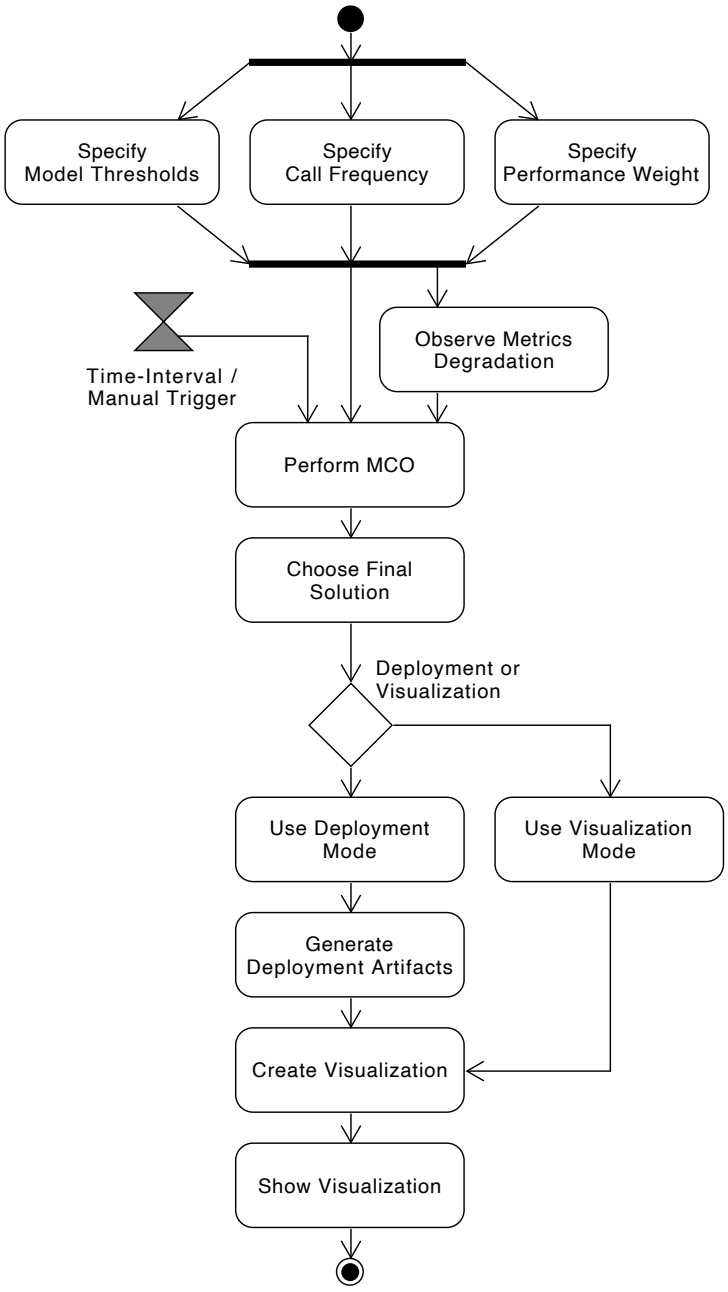


Figure 5.3.: Model Reconfiguration Toolflow

tool generates deployment artifacts. The visualization mode skips this step. In both modes, a PlantUML visualization is created and shown.

Figure 5.3 shows the flow regarding the model reconfiguration. Users specify model thresholds, call frequency of client requests, and performance weight. A reconfiguration is triggered when metrics degradation is observed, according to timers or manually. When reconfiguration is triggered, the backend performs an MCO analysis and chooses a final reconfiguration solution. If the deployment mode is chosen, deployment artifacts will be generated. The reconfiguration visualization is then created and shown.

5.4. Evaluation

In this section, we evaluate our architecture by comparing the reliability and performance predictions to the empirical results of our experiment. Note that the ADR architecture is neither specific to our experiment infrastructure nor to our cases. We use our empirical data set in the online artifact of this doctoral thesis[6] for the evaluation of our approach using measured empirical data (see Section 2.3 for experiment details).

5.4.1. Evaluation Cases

We systematically evaluate our proposed architecture through various thresholds and importance weights for reliability and performance. We compare our model predictions with our 36 experiment cases given below. That is, we compare our results with three levels of services, three fixed architecture configurations, and four levels of call frequency:

$$n_{serv} \in \{\ 3, 5, 10\ \} \tag{5.22}$$

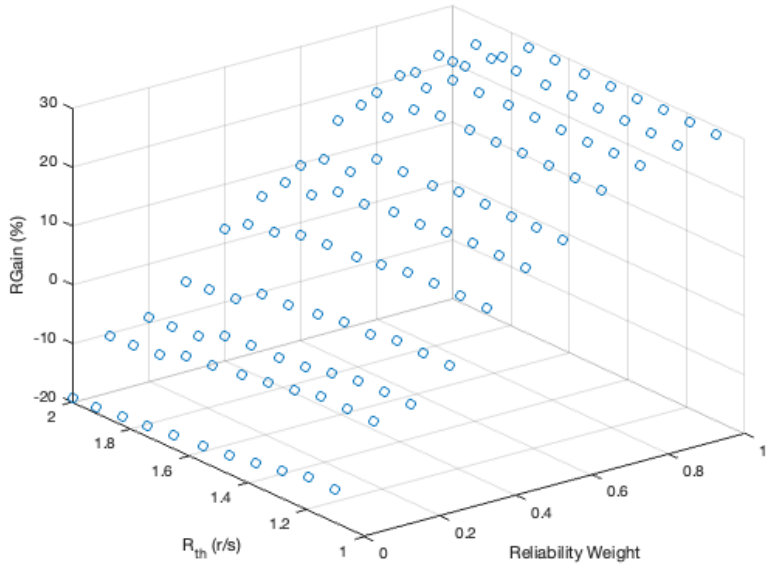
$$n_{rout} \in \{\ 1, 3, n_{serv}\ \} \tag{5.23}$$

$$cf \in \{\ 10, 25, 50, 100\ \}\ r/s \tag{5.24}$$

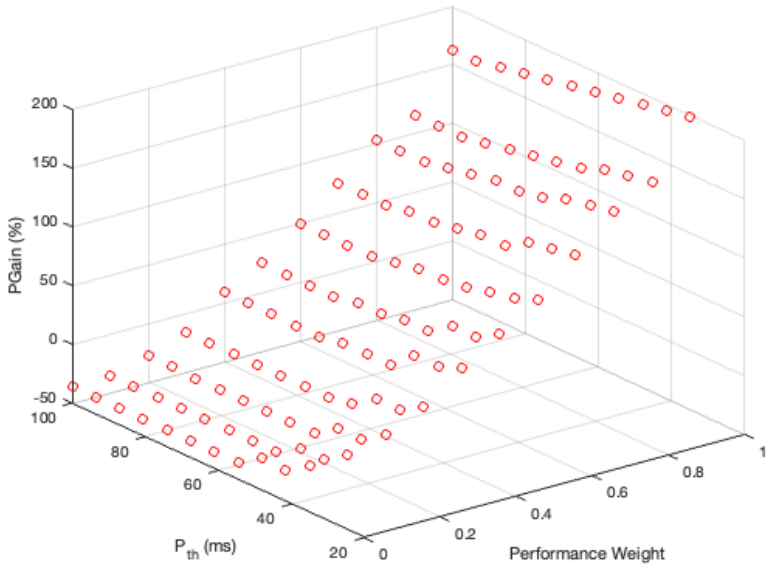
Regarding reliability and performance thresholds, we start with very tight reliability and very loose performance thresholds so that only centralized routing is acceptable. We increase the reliability and decrease the performance thresholds by 10% in each step so that distributed routing becomes applicable. To find the starting points, we consider the worst-case scenario of our empirical data. A higher n_{serv} results in a higher expected request loss (as analyzed in Chapter 3). In our experiment, the highest number of services is ten. With $n_{serv} = 10$, the worst-case reliability for centralized routing and completely distributed routing ($n_{rout} = 10$) is 1.1 and 2.0 r/s, respectively. Regarding performance, for the case of $n_{serv} = 10$, we investigate our predictions to find a range where a reconfiguration is possible. The lowest possible performance prediction is 33.7 ms, and the highest is 101.1 ms. We adjust these values slightly and take our boundary thresholds as:

$$1.1 \leq R_{th} \leq 2.0\ r/s \tag{5.25}$$

$$35 \leq P_{th} \leq 100\ ms \tag{5.26}$$



(a) Reliability Gain



(b) Performance Gain

Figure 5.4.: Reliability and Performance Gains Compared to Fixed Architecture Configurations (each point is an average of 36 experiment cases.)

We start with an importance weight of 1.0 for reliability and 0.0 for performance. We decrease the reliability importance and increase the performance weight by 10% in each iteration. We evaluate 4356 systematic evaluation cases: 36 experiment cases, 11 importance weight levels, and 11 thresholds. To support reproducibility, the evaluation script and log are provided in the online artifact of this dissertation[6].

5.4.2. Results Analysis

We define reliability gain, i.e., $RGain$, and performance gain, i.e., $PGain$, as the average percentage differences of our predictions compared to those of fixed architectures. Let $Cases$ be the experiment cases and n_c the length of $Cases$. Note that the gains are based on the Mean Absolute Percentage Error (MAPE) widely used in the cloud quality-of-service research [90].

$$RGain = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{R_c - R_{n_{rout}}}{R_{n_{rout}}} \tag{5.27}$$

$$PGain = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{P_c - P_{n_{rout}}}{P_{n_{rout}}} \tag{5.28}$$

Remember $R_{n_{rout}}$ and $P_{n_{rout}}$ are reliability and performance predictions (see the MCO analysis in Section 5.2). We have 36 experiment cases and $n_c = 36$ in our evaluation.

Figure 5.4 shows the reliability and performance gains compared to the predictions of fixed architecture configurations, i.e., when there are no adaptations. Our adaptive architecture provides improvements in both reliability and performance gains. As more importance is given to the reliability of a system, i.e., reliability weight increases, our architecture reconfigures the routers so that the gain in reliability rises as shown by Figure 5.4a. Regarding performance, the same trend can be seen in Figure 5.4b. A higher performance weight results in a higher performance gain. On average, when cases with correct and incorrect architecture choices are analyzed together, the ADR architecture provides 9.82% and 47.86% reliability and performance gains, respectively. A higher gain for performance compared to reliability is expected. To clarify, studying Equations (5.3) and (5.4) informs that changing the number of routers has a higher effect on the performance than a system's reliability. In Chapter 4, we defined performance as the average processing time of requests per router. Having a higher number of routers to process the requests in parallel results in dividing the average processing time by more routers. However, the sum of the number of services and routers affects the reliability.

5.5. Threats to Validity

There are several threats to the validity and limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [96].

5.5.1. Construct Validity

The accurate representation of the intended construct by a measurement is assessed through construct validity. We used request loss and the average processing time of requests per router as reliability and performance metrics, respectively. While this is a common approach in service- and cloud-based research (see Section 1.3), the threat remains that other metrics might model these quality attributes better, e.g., a cascade of calls beyond a single call sequence for reliability [67], or data transfer rates of messages which are m byte-long for performance [53]. More research, probably with real-world systems, is required for this threat to be excluded.

5.5.2. Internal Validity

Internal validity concerns factors that affect the independent variables concerning causality. Dynamic routing architectures are based on many different technologies. Our ADR architecture abstracts the controlling logic component in dynamic routing under a router concept to allow interoperability between these architectural patterns. In a real-world system, changing between these technologies is not always an easy task, but it is not impossible either. In this chapter, we provided a scientific proof-of-concept based on an experiment with the prototypical implementation of these technologies. The threat remains that changing between these technologies in a real-world application might have other impacts on reliability and performance, e.g., network latency increasing processing time.

5.5.3. External Validity

External validity concerns threats that limit the ability to generalize the results beyond the experiment. We designed our novel architecture with generality in mind. However, the threat remains that evaluating our approach based on another infrastructure may lead to different results. To mitigate this thread, we systematically evaluated the proposed architecture with 4356 evaluation cases using the data of our extensive experiment of 1200 hours (see Section 5.4). Moreover, the results might not be generalizable beyond the given experiment cases of 10-100 requests per second and call sequences of length 3-10. As this covers a wide variety of loads and call sequences in cloud-based applications, the impact of this threat should be limited.

5.5.4. Conclusion Validity

Conclusion validity concerns factors that affect the ability to conclude the relations between treatments and study outcomes. As the statistical method to evaluate the accuracy of our model's predictions, we defined reliability and performance gains based on the Mean Absolute Percentage Error (MAPE) metric [90] as it is widely used and offers good interpretability in our research context.

5.6. Conclusions

In this chapter, we studied a multi-criteria optimization analysis to choose an optimal solution in service- and cloud-based applications when reliability and performance are considered. We hypothesized that a self-adaptation between dynamic routing architectures is beneficial over fixed architecture selections. We proposed a routing architecture that dynamically self-adapts between different routing patterns based on the need of an application. Moreover, we systematically evaluated our approach using 4356 evaluation cases based on the empirical data of our scientific experiment. The results confirm our hypothesis that the ADR architecture can adapt the routing pattern in a running system to optimize reliability and performance. Even on average, where cases with the right and the wrong architecture choices are analyzed together, our approach offers a 9.82% reliability gain and a 47.86% performance gain.

To the best of our knowledge, no architecture has been presented in the literature that dynamically adjusts reliability and performance trade-offs, specifically in service- and cloud-based dynamic routing. Our architecture adapts, based on triggers, e.g., change of incoming load frequency or degradation of monitoring data, to an optimal configuration to prevent request loss or an increase in process times of requests. Before our work, architects needed to redesign and redeploy architecture configurations manually.

6. Multidimensional Autoscaling

This chapter addresses the research problem P_4: *Lack of an approach to autoscale components multidimensionally to prevent overload*. We model the system components as queuing stations and study them when overloaded. We investigate horizontal autoscaling, i.e., adding replicas, and vertical autoscaling, i.e., adding resources to a component. In Chapter 8, we reuse the approach presented here for a multifaceted reconfiguration of dynamic routing applications to address the research problem P_6: *Lack of tool support for the multifaceted reconfiguration of dynamic routing applications*.

6.1. Introduction

Dynamic reconfiguration is commonly used to accommodate the dynamic behavior of today's applications. As cloud-based systems become increasingly complex, it is hard and cost-ineffective to manage them manually. Dynamic routers, such as API gateways [79] or message brokers [45], in combination with autoscalers can adapt the system to the resource demands, e.g., when a sudden load spike for a specific part of the system is observed. Without taking the costs of cloud resources into account, this reconfiguration can lead to a significant increase in charges. In this thesis, we propose a self-adaptive and cost-aware Adaptive Dynamic Routers (ADR) architecture. Our approach in this chapter performs a Multi-Criteria Optimization (MCO) analysis [4] to automatically reconfigure the routers and the services of a cloud-based system, considering the reconfiguration costs. This multidimensional autoscaling of resources takes incoming load as an input and uses queuing theory [51] to find an optimal reconfiguration solution. We systematically evaluated our architecture with an extensive number of evaluation cases (9600). On average, over cases where an overload is predicted, our approach reduces the overload rate by 46.7% and 61.8% for routers and services, respectively.

The structure of the article is as follows. Section 6.2 gives an overview of the approach we follow in this chapter. Section 6.3 explains the approach in detail. We study the parameterization of our models in Section 6.4 and provide an illustrative sample case in Section 6.5. Section 6.6 presents the evaluation of our research. We discuss the threats to the validity in Section 6.7 and conclude in Section 6.8.

6.2. Approach Overview

In this section, we present the overview of our approach to model and prevent system overload. Table 6.1 presents the mathematical notations used in this chapter.

Table 6.1.: The Mathematical Notations Used in this Chapter

Notation	Description
l	Length of the number of requests a buffer can store
μ	Processing rate
μ_i	Processing rate of a reconfigurable component i
μ_{pro}	Increased processing rate of a component
λ	Arrival rate
λ_i	Arrival rate of a reconfigurable component i
r/s	Requests per second
HAS	Horizontal autoscaling
VAS	Vertical autoscaling
FS	Full state of a buffer
FS_{th}	Full state threshold
IR_i	Number of incoming requests for a reconfigurable component i
BFR	Buffer fill rate
BFR_i	Buffer fill rate of a reconfigurable component i
$\overline{\Delta BFR}$	Average percentage difference of buffer fill rate
$BFR(n_{scal}, n_{pro})$	Predicted buffer fill rate for n_{scal} and n_{pro}
n_{scal}	Number of scaling replicas
n_{pro}	Processing rate improvements
cf	Call frequency
cf_{scal}	Call frequency after scaling-out a component
$C(n_{scal})$	Cost of scaling-out the component by n_{scal} replicas
$C(n_{pro})$	Cost of increasing the processing rate by n_{pro} r/s
$C(n_{scal}, n_{pro})$	Reconfiguration cost for n_{scal} and n_{pro}
C_{th}	Cost threshold
$\overline{C}$	Average reconfiguration cost
RR	Reconfiguration ratio
$RR(n_{scal}, n_{pro})$	Reconfiguration ratio for n_{scal} and n_{pro}
n_{rout}	Number of routers
n_{serv}	Number of services
σ	Standard deviation of the empirical data
Q_1	First quartile of the empirical data
Q_3	Third quartile of the empirical data
$Cases$	Set of the number of services and the number of routers
n_c	Length of $Cases$

6.2.1. Architecture Extensions

We require a couple of significant extensions to the ADR architecture for our approach presented in this chapter, i.e., to prevent system overload. Firstly, the configurator

components monitor and reconfigure the router and the services of an application. As shown in Figure 2.4, the *reconfigurable components* refer to services and routers collectively. Secondly, we perform the reconfiguration to prevent system overload. To do so, we model reconfigurable components as queuing stations and use queuing theory [51]. We identify system overload when any reconfigurable system component overloads, resulting in an application being non-responsive to client requests.

Thirdly, the perspective for the reconfiguration is different in this chapter. The proposed architecture focuses on each reconfigurable component, i.e., a router or a service, separately and reconfigures it individually to prevent the overload of that specific component. As a result, we study reconfiguration measures we did not investigate before, such as autoscaling. Finally, we consider the reconfiguration cost as a deciding factor, i.e., an optimization criterion.

6.2.2. Reconfigurable Components as Queuing Stations

We model each reconfigurable component, i.e., a router or a service, as a queuing station having two subcomponents, namely a buffer and a processor, as shown in Figure 6.1. Incoming requests are buffered in a queue and processed by the processor one by one according to a queuing discipline, e.g., a first-come-first-served strategy. Let l be the length of the number of requests a buffer can store, and μ the processing rate of a processor based on the number of requests per second r/s. We characterize a queuing station by l of its buffer and μ of its processor. Many cloud providers offer a standard service to configure containers. For instance, Google Kubernetes Engine Autopilot[1] allows the configuration of vCPUs and memories per containers. The same service is offered by Microsoft Azure Container Instances[2] and Amazon Elastic Container Service[3] on Fargate[4].

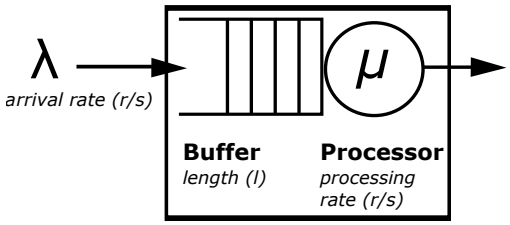


Figure 6.1.: Components as Queuing Stations

We indicate a system overload when a buffer of a reconfigurable component overloads. That is when a component as a queuing station is not in its *steady state* [51]. Let μ_i and λ_i be the processing and arrival rates of a reconfigurable component i, respectively. For a reconfigurable component i to be in a steady state, its processing rate must be greater than or equal to its arrival rate:

$$\mu_i \geq \lambda_i \tag{6.1}$$

[1] https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview
[2] https://azure.microsoft.com/en-us/services/container-instances/
[3] https://aws.amazon.com/ecs/
[4] https://aws.amazon.com/fargate/

When $\mu_i < \lambda_i$, the buffer of a reconfigurable component i eventually overloads which, in turn, results in a system overload. For the sake of simplicity, we only consider homogeneous workload, i.e., single class requests.

6.2.3. Measures to Overcome System Overload

When a system overload is predicted (see Section 6.3.1) for a reconfigurable component, i.e., a router or a service, we consider the following measures that can be taken to address system overload at the component level:

- Scale out the reconfigurable component using replicas, i.e., *horizontal autoscaling* HAS.
- Increase the processing rate of the reconfigurable component, i.e., *vertical autoscaling* VAS.
- Increase the length of the component's buffer.

Note that this list is not exhaustive, i.e., many different measures can be taken, e.g., using queues with bounded waiting time [58] in which requests will be dropped if the buffer overloads, using a circuit breaker [78], changing the router technology, e.g., using an event streaming platform [79], or adding more routers and reconfiguring the routing as presented in Chapter 5.

We can consider the above measures from a high level of abstraction, i.e., at the component level. Note that if a reconfigurable component is overloading, increasing the buffer length only slows down the process and does not prevent system overload. Moreover, the architect can set the buffer length from the beginning to a pre-defined maximum. Therefore, we specifically study the following measures:

- Scale out an overloading router using replicas, i.e., HAS.
- Scale out an overloading service using replicas, i.e., HAS.
- Increase the processing rate of a router, i.e., VAS.
- Increase the processing rate of a service, i.e., VAS.

These measures are associated with cloud costs since they add cloud resources to the system (see Section 6.4.4). In the remainder of this chapter, we define an analytical model of system overload and use this model to automatically adjust the cloud resource usage considering the cost of cloud deployment. Note that the ADR architecture considers each component, i.e., a router or a service, separately and performs the reconfiguration individually, hence the differentiation in the above list.

6.3. Approach Details

In this section, we first introduce our system overload model, then investigate cloud-based cost models in relation to our model, and present our automatic reconfiguration algorithm.

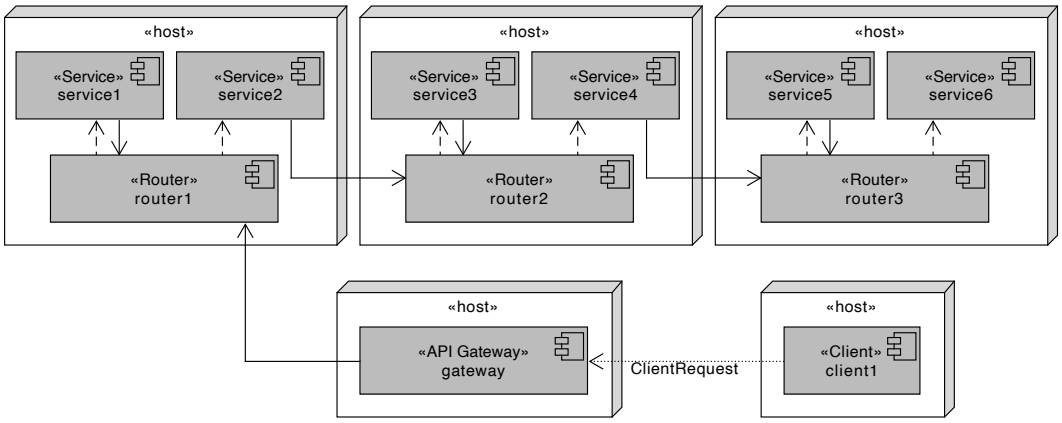


Figure 6.2.: Example Configuration with Incoming Requests for Routers (Solid Lines) and Services (Dashed Lines)

6.3.1. System Overload Model

The QoS monitor observes each reconfigurable component, i.e., a router or a service, and triggers a reconfiguration for each component separately if required (see Figure 2.5). To do so, it monitors the number of requests each component receives, i.e., its arrival rate. The monitor observes the processing rate of all routers and services. This can be achieved, e.g., by executing a `docker stats` command when Docker[5] containerization is used to read the CPU usage of the container. The QoS monitor can predict a system overload if an arrival rate of a reconfigurable component is higher than its processing rate and its buffer is alarmingly full.

Arrival Rate of Reconfigurable Components

To model the arrival rate λ_i of each reconfigurable component i, we define the incoming requests from the viewpoint of a component. That is the requests received by a component and stored in its buffer. An example configuration is presented in Figure 6.2. Solid lines represent the incoming requests of routers, and dashed lines show those of services. Note that this figure shows the incoming requests per one client request.

We define call frequency cf as the frequency with which the API gateway receives the client requests based on requests per second r/s. Let IR_i be the number of incoming requests for a reconfigurable component i:

$$\lambda_i = cf \cdot IR_i \tag{6.2}$$

The arrival rate for a reconfigurable component i is the call frequency multiplied by the number of incoming requests IR_i. To illustrate, in the example configuration, $IR_i = 2$

[5] https://www.docker.com

uniformly for all routers. When the application is under stress with, e.g., $cf = 10\ r/s$, each router has an arrival rate of $\lambda_i = 20\ r/s$. Note that IR_i needs to be parameterized for each application separately. Section 6.4 presents an example parameterization.

Buffer Fill Rate

Let BFR_i be the buffer fill rate of a reconfigurable component i. We define BFR_i as the difference between the arrival and processing rates:

$$BFR_i = \lambda_i - \mu_i \tag{6.3}$$

When the buffer fill rate is positive for any reconfigurable component, i.e., a router or a service, the arrival rate of the component is higher than its processing rate. In this case, the system eventually overloads if the call frequency cf of client requests does not decrease. Using Equation (6.2), we can rewrite Equation (6.3) as:

$$BFR_i = cf \cdot IR_i - \mu_i \tag{6.4}$$

6.3.2. Threshold for Reconfiguration

Let FS be the full state of a buffer indicating how full a buffer is. The severity of the damage of a positive buffer fill rate depends on FS:

$$0.0 < FS < 1.0 \tag{6.5}$$

Let FS_{th} be the full-state threshold. An architect can define an FS_{th} for the buffers of the reconfigurable components of a system. If any buffer reaches this threshold, the QoS monitor triggers the manager to reconfigure the architecture (see Figure 2.5). In such a case, the manager reduces the buffer fill rate BFR_i of a reconfigurable component i by performing a combination of the following measures:

- Scale out a reconfigurable component i by using replicas reducing its arrival rate λ_i.
- Increase the processing rate μ_i of a reconfigurable component i to reduce its BFR_i (see Equation (6.3)).

Conversely, if the FS of a buffer goes below the threshold, the manager can reverse the steps taken, e.g., scale in the component replicas.

6.3.3. Reconfiguration Algorithm

The proposed architecture automatically reconfigures the system at run-time using an MCO analysis [4]. We consider the following optimization criteria: The buffer fill rate prediction and the reconfiguration costs.

Prediction of Buffer Fill Rate

The manager uses the buffer fill rate predictions to decide how much of each reconfiguration measure to perform. Remind that the architecture monitors each reconfigurable component separately and reconfigures it individually. Let $BFR(n_{scal}, n_{pro})$ be the predicted buffer fill rate by improving n units in each reconfiguration measure, i.e., scaling out a component by n_{scal} replicas, or improving the processing rate of an alarming component by n_{pro} requests per second r/s.

$$BFR(n_{scal}, n_{pro}) = cf_{scal} \cdot IR_i - \mu_{pro} \tag{6.6}$$

IR_i is the number of incoming requests of a reconfigurable component i, cf_{scal} is the call frequency after scaling out the component and load balancing the cf among the replicas, and μ_{pro} is the increased processing rate of the component:

$$\mu_{pro} = \mu_i + n_{pro} \tag{6.7}$$

That is, the processing rate μ_i of the reconfigurable component i is increased by n_{pro} r/s.

Cost of the Reconfiguration Measures

We consider the cost models of widely-used cloud providers so that our proposed approach can be applied to real-world applications. These cost models relate to resource usage and are at the container level. That is, customers pay per use of resources their containers need. For instance, Google Kubernetes Engine Autopilot[6] and Microsoft Azure Container Instances[7] charge per use of vCPUs and memories in seconds from the time the container images are pulled until the task is finished and there is no minimum cost. However, Amazon Fargate[8] charges a minimum of one minute. Afterward, the cost is calculated per second.

The reconfiguration measures mentioned in Section 6.2.3 require an increase in cloud resource usage. We associate costs with improving n units for each reconfiguration measure. Let $C(n_{scal}, n_{pro})$ be the reconfiguration cost:

$$C(n_{scal}, n_{pro}) = C(n_{scal}) + C(n_{pro}) \tag{6.8}$$

$C(n_{scal})$ associates with the cost of scaling out a component by n_{scal} replicas, and $C(n_{pro})$ specifies the cost of increasing the processing rate by n_{pro} requests per second r/s.

Note that the conversion between vCPU and processing rate μ_i depends on the application. There are different CPU types with different processing capabilities (see, e.g., Google Cloud CPU Platforms[9]). Moreover, the requests for each application have different timing needs. We provide a systematic evaluation in Section 6.6. Also, consider that the cost model is general and is not specific to these cloud providers. Other cloud cost models can be used if necessary (see Section 6.4 for an example specification).

[6] https://cloud.google.com/kubernetes-engine/pricing
[7] https://azure.microsoft.com/en-us/pricing/details/container-instances/
[8] https://aws.amazon.com/fargate/pricing/
[9] https://cloud.google.com/compute/docs/cpu-platforms

Multi-Criteria Optimization

The QoS monitor triggers the manager to reconfigure the components (see Figure 2.5). When the full state threshold FS_{th} has reached for a reconfigurable component, the configurator automatically adapts the buffer fill rate BFR of the component based on an MCO analysis [4]. Consider the following optimization problem: The configurator must decide how many units of each measure to improve to have the minimum predicted buffer fill rate. However, simultaneously, the configurator needs to minimize the cost of cloud resource usage. Let C_{th} be the cost threshold:

$$\begin{aligned} &Minimize \\ &\quad BFR(n_{scal}, n_{pro}) && (6.9) \\ &\quad C(n_{scal}, n_{pro}) && (6.10) \\ &Subject\ to \\ &\quad C(n_{scal}, n_{pro}) \leq C_{th} && (6.11) \end{aligned}$$

Note that any number of constraints can be added. The objective functions oppose each other: We want to minimize the buffer fill rate but at the minimum reconfiguration cost. Typically, there is no single answer to an MCO problem but a set of acceptable points in the solution space called the Pareto front [4]. Section 6.5 provides an illustrative sample case.

Preference Function

The final reconfiguration choice from the Pareto front is upon the decision maker based on their preference. We define a preference function so the manager can choose a point as the final solution automatically. Let RR be the reconfiguration ratio, i.e., the ratio of buffer-fill-rate improvements to the reconfiguration costs. We select a final solution based on the $RR(n_{scal}, n_{pro})$, i.e., the reconfiguration ratio of a solution with n_{scal} scaling replicas and n_{pro} r/s processing rate improvements:

$$RR(n_{scal}, n_{pro}) = \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{C(n_{scal}, n_{pro})} \tag{6.12}$$

This gives us the buffer fill rate improvements for each unit of cost spent on the reconfiguration. The preference function goes through all points on the Pareto front and chooses the solution with the highest RR.

Automatic Reconfiguration

The component QoS monitor observes all components, i.e., routers and services, of the system (see Section 2.2). The monitor triggers the manager whenever a system overload is predicted for a component. This means whenever a component's arrival rate λ is higher than its processing rate μ, and its buffer is alarmingly full (indicated by the full state FS of the buffer). In this case, the manager uses the reconfiguration algorithm presented in Algorithm 2 to deploy new components or reconfigure the existing ones.

Algorithm 2: Reconfiguration Algorithm for an Overloading Component

Input: $C(n_{scal}), C(n_{pro}), IR$

$cf, \mu \leftarrow$ **consumeMonitoringData()**

paretoFront $\leftarrow$ **MCO**(cf, IR, μ)

reconfigSolution $\leftarrow$ **preferenceFunction**(paretoFront)

reconfigure(reconfigSolution)

function preferenceFunction(paretoFront)
begin
  $RR \leftarrow 0$
  reconfigSolution $\leftarrow (0, 0)$
  foreach *solution : paretoFront* **do**
    $RR(n_{scal}, n_{pro}) \leftarrow \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{C(n_{scal}, n_{pro})}$
    if $RR(n_{scal}, n_{pro}) > RR$ **then**
      $RR \leftarrow RR(n_{scal}, n_{pro})$
      reconfigSolution $\leftarrow (n_{scal}, n_{pro})$
    end
  end
  return reconfigSolution
end

6.4. Parameterization of Model to Experiment Parameter Values

In Section 6.3, we introduced our proposed system overload model, which is general for dynamic routing architectures. Some model elements need to be parameterized based on the specific application. Namely, the incoming requests IR_i for a reconfigurable component i must be specified. Additionally, the call frequency after scaling out a component and load-balancing the call frequency cf among the scaling replicas, i.e., cf_{scal}, must be specified. Moreover, the cost functions must be parameterized based on the ADR application.

6.4.1. Incoming Requests

Figure 6.2 shows an example configuration of our experiment. As can be seen, the number of incoming requests is different for routers and services in our experiment. For **services**, we can easily observe that there is one incoming request for each service:

$$IR_i = 1 \tag{6.13}$$

We parameterize the incoming requests for **routers** in our experiment. Let n_{serv} be the number of services and n_{rout} the number of routers of an ADR application. In

Section 2.3.2, we mentioned that we distributed services equally among routers:

$$IR_i = \frac{n_{serv}}{n_{rout}} \tag{6.14}$$

In Figure 6.2, we have six services and three routers; consequently, we have a uniform $IR_i = 2$ for all routers.

6.4.2. Predicted Call Frequency

When scaling out an ADR application with n_{scal} replicas, we must know the load-balancing strategy. For the sake of simplicity, we consider an equal load balancing, i.e., an equal distribution of the call frequency.

$$cf_{scal} = \frac{cf}{n_{scal} + 1} \tag{6.15}$$

Note that the QoS monitor observes all components, i.e., routers and services, and triggers the reconfiguration for each component individually. For each alarming component, we have one instance with n_{scal} replicas after this reconfiguration measure is performed. As a result, we have $n_{scal} + 1$ replicas.

6.4.3. Predicted Buffer Fill Rate

We can parameterize the predicted buffer fill rate presented by Equation (6.6) using Equations (6.7), (6.13) and (6.15) for a service i:

$$BFR(n_{scal}, n_{pro}) = \frac{cf}{n_{scal} + 1} - (\mu_i + n_{pro}) \tag{6.16}$$

Using Equations (6.7), (6.14) and (6.15), we have the following for a router i:

$$BFR(n_{scal}, n_{pro}) = \frac{n_{serv}}{n_{rout}} \cdot \frac{cf}{n_{scal} + 1} - (\mu_i + n_{pro}) \tag{6.17}$$

6.4.4. Cost Functions

As explained before, our model can easily be combined with the cost functions of different cloud providers. To illustrate this, we show one example by mapping our observed processing rates to an existing cost model, i.e., the Google Autopilot pricing[10]. This can easily be adapted to the related cost models such as the one of Amazon Fargate[11]. Note that for very different cost models, a new cost function needs to be defined.

Cloud costs relate to the resource demand of an application. An architect must map the processing rate in requests per second r/s to vCPU usage. Since the focus of the ADR architecture is on architecting the system and studying different configurations,

[10] https://cloud.google.com/kubernetes-engine/pricing
[11] https://aws.amazon.com/fargate/pricing/

an estimate is sufficient. We used our experiment infrastructure for this estimation (see Section 2.3 for details). To do so, we ran different configurations, i.e., multiple incoming load profiles, and investigated the CPU usage of the containers. Since we use Docker technology[3], the command `docker stats` gives information regarding CPU usage. In our experiment, a 100% vCPU usage relates to a processing rate of about 32 r/s.

Multiple cloud providers allow increments of 0.25 vCPUs per container, e.g., Google Autopilot or Amazon Fargate. Therefore, we consider increments of 0.25 vCPUs corresponding to 8 r/s, which costs roughly according to Google Autopilot pricing[10] with maximum memory:

$$C(vCPU = 0.25) = 5 \cdot 10^{-4}\ cents\,/s \tag{6.18}$$

For instance, if a container needs a processing power of 2 vCPUs corresponding to processing rate $\mu = 64\ r/s$:

$$C(n_{scal} = 1) = 8 * C(vCPU = 0.25) = 4 \cdot 10^{-3}\ cents\,/s \tag{6.19}$$

For $C(n_{scal})$, i.e., the reconfiguration costs of scaling out the components by n_{scal} replicas, we multiply the number of replicas by the cost of the component:

$$C(n_{scal}) = n_{scal} \cdot C(n_{scal} = 1) \tag{6.20}$$

6.5. Illustrative Sample Case

This section presents an illustrative example to demonstrate the concepts of this study.

6.5.1. Multi-Criteria Optimization

Let us consider the following example: An ADR application with six services and three routers (see Figure 6.2) is under stress with a call frequency of $cf = 100$ r/s. The QoS monitor asserts that the full state threshold $FS_{th} = 0.6$ has reached for the buffer of a router i, resulting in a reconfiguration trigger (see Section 6.3.2). The processing rate of the router i is $\mu_i = 64$ r/s corresponding to 2 vCPUs (see Section 6.4.4):

$$n_{serv} = 6 \tag{6.21}$$
$$n_{rout} = 3 \tag{6.22}$$
$$\mu_i = 64.0\ r/s \tag{6.23}$$
$$cf = 100.0\ r/s \tag{6.24}$$
$$FS_{th} = 0.6 \tag{6.25}$$

We can calculate the number of incoming requests for the router i according to Equation (6.14) as:

$$IR_i = \frac{n_{serv}}{n_{rout}} = 2.0 \tag{6.26}$$

Using Equation (6.2), the arrival rate of the router i is:

$$\lambda_i = cf \cdot IR_i = 200.0\ r/s \tag{6.27}$$

Therefore, the buffer fill rate of the router i can be calculated using Equation (6.3):

$$BFR_i = \lambda_i - \mu_i = 136.0\ r/s \tag{6.28}$$

Assume an architect has chosen a cost threshold of $C_{th} = 1\ cents/s$ for the router i. We can rewrite the MCO analysis in Equations (6.9) and (6.10) using Equations (6.16) and (6.20):

$$Minimize$$

$$2 \cdot \frac{100}{n_{scal} + 1} - (n_{pro} + 80) \tag{6.29}$$

$$8 \cdot n_{pro} \cdot C(vCPU = 0.25) + n_{scal} \cdot C(n_{scal} = 1) \tag{6.30}$$

$$Subject\ to$$

$$8 \cdot n_{pro} \cdot C(vCPU = 0.25) + n_{scal} \cdot C(n_{scal} = 1) \leq 1\ cents/s \tag{6.31}$$

Remember that we consider increments of 0.25 vCPUs corresponding to 8 r/s.

MCO Solution Space

Table 6.2 presents the MCO solution space of the illustrative sample case, as well as the reconfiguration ratio RR for each solution. Note that we only consider the cases where the buffer fill rate BFR reduces to a minimum of zero to have a bound on the amount of resource usage. That is, we add enough resources to prevent the overload of a component and not more. Other strategies could be easily added by an architect based on the need of an application, e.g., reducing the BFR to a defined sub-zero minimum.

Table 6.2.: MCO Solution Space of the Illustrative Sample Case

n_{pro}	n_{scal}	$BFR(n_{scal}, n_{pro})$	$C(n_{scal}, n_{pro})$	RR
0.0	0.0	136.0	0.0	0.0
40.0	0.0	96.0	16.0	2.5
80.0	0.0	56.0	32.0	2.5
0.0	1.0	36.0	40.0	2.5
120.0	0.0	16.0	48.0	2.5
0.0	2.0	3.0	80.0	2.0

6.5.2. Final Reconfiguration Solution

In Section 6.3.3, we mentioned that the preference function goes through all points on the Pareto front and returns the solution with the highest reconfiguration ratio RR. In Table 6.2, the final solution is:

$$(n_{pro}, n_{scal}) = (40, 0) \tag{6.32}$$

which gives the highest reconfiguration ratio, i.e., $RR = 2.5$, with the lowest reconfiguration cost. That is, by spending 16 $cents/s$ the buffer fill rate BFR_i of the router i reduces from 136 r/s (calculated by Equation (6.28)) to 96 r/s. Note that an architect can select a different strategy as well, e.g., choosing the lowest BFR as the final solution, in which case the more costly $(n_{pro}, n_{scal}) = (120, 0)$ is the answer.

6.6. Evaluation

In this section, we evaluate our approach by systematically calculating the buffer full rate improvements of an extensive number of 9600 evaluation cases. The evaluation script, the calculated Pareto fronts, and the evaluation log containing detailed information of each systematic evaluation case are anonymously downloadable to support reproducibility in the online artifact of this thesis[6]. The ADR architecture provides a general approach and needs to be parameterized based on the application for which it is applied. We use our experiments reported in Section 2.3.2 for this parameterization. Note that the proposed architecture is not specific to our experiment infrastructure. Architects can easily tailor ADR to their specific use case, as we explain in this section for our experiment.

To systematically evaluate our approach, we use our experiment cases and go through a range of values.

$$n_{serv} \in (3, 5, 10) \tag{6.33}$$

$$n_{rout} \in (1, 3, n_{serv}) = (1, 3, 5, 10) \tag{6.34}$$

In our experiment, we considered the call frequency cf between 10 and 100 r/s. Since we are studying system overload in this chapter, we consider higher loads. That is, we consider 20 levels of the call frequency cf in the following range, each step increasing 10 r/s:

$$10 \leq cf \leq 200\ r/s \tag{6.35}$$

For container configurations, we investigate 20 levels based on the vCPU requirements of a container between 0.25 and 5 vCPUs. We increase 0.25 vCPUs in each level incrementally. As explained in Section 6.4.4, 0.25 vCPUs correspond to a processing rate of $\mu = 8\ r/s$ in our experiment:

$$8 \leq \mu \leq 160\ r/s \tag{6.36}$$

Regarding the cost threshold, we take $C_{th} = 1$ cent per second. Note that this threshold is considered for each reconfiguration step and each component separately. All in all, we

performed an extensive evaluation of 9600 cases, i.e., three levels of n_{serv}, four levels of n_{rout}, twenty levels of cf, and twenty levels of μ, which are all considered for routers and services separately. Let $\overline{C}$ be the average reconfiguration cost, $\overline{\Delta BFR}$ the average percentage difference of buffer fill rate, $Cases$ the set of the number of services and the number of routers, and n_c the length of $Cases$.

$$\overline{\Delta BFR} = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{BFR(0,0)} \tag{6.37}$$

$$\overline{C} = \frac{1}{n_c} \cdot \sum_{c \in Cases} C(n_{scal}, n_{pro}) \tag{6.38}$$

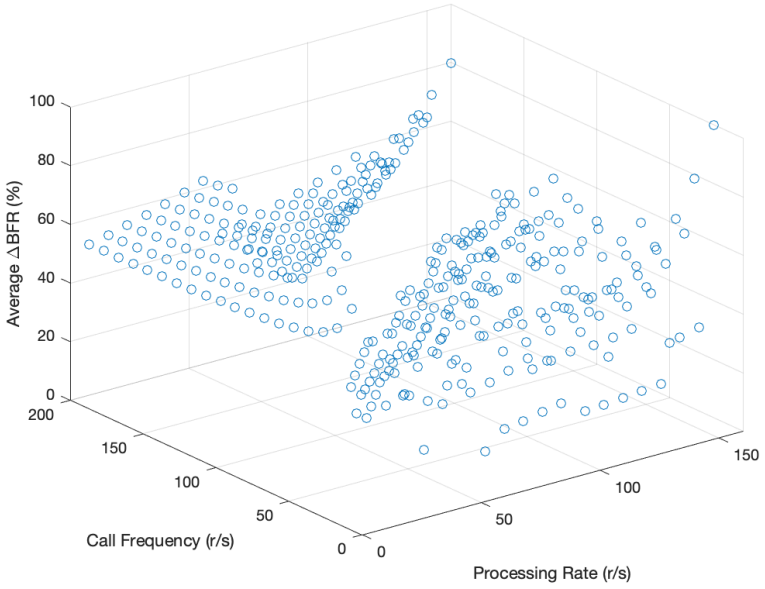
Note that $\overline{\Delta BFR}$ is based on the Mean Absolute Percentage Error (MAPE) metric [90]. $Cases$ is the set of $n_{serv} \in (3,5,10) \times n_{rout} \in (1,3,5,10)$ and $n_c = 12$.

Figure 6.3 shows the evaluation data regarding the average percentage difference of buffer fill rate. We can see that as the processing rate μ of a container increases, i.e., the number of vCPUs per container rises, we have a higher $\overline{\Delta BFR}$, especially with a higher call frequency cf. The ADR architecture yields the most improvements when a container with a high processing rate is under the stress of a high load. Figure 6.4 presents the evaluation data regarding the average reconfiguration cost. Regarding $\overline{C}$ for routers, as shown in Figure 6.4a, the reconfiguration costs rise as the processing rate μ of router containers increases. This is expected because, e.g., scaling out a router with five vCPUs costs higher than scaling out a router with only one vCPU. However, for service containers with higher processing power, there is a constant average cost, as shown in Figure 6.4b. In these cases, our approach chooses the same reconfiguration measure, i.e., adding a processing rate of $\mu = 40\ r/s$ to an overloading service. This is because as the processing power of a service container increases, the cost of scaling out the service also rises. In this case, adding 40 r/s with an average cost of 0.0016 cents per second gives the highest reconfiguration ratio RR and is chosen repeatedly.

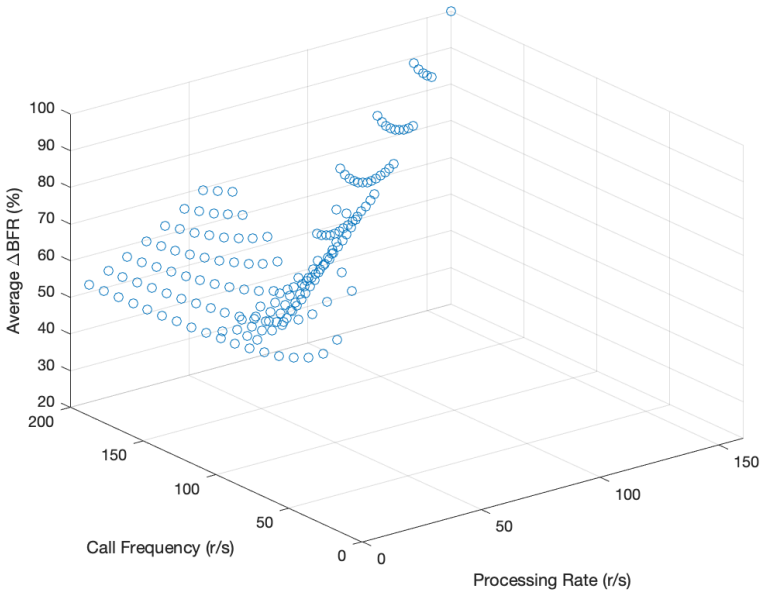
Table 6.3 reports the statistics of $\overline{\Delta BFR}$ and $\overline{C}$, in which σ, Q_1 and Q_3 are the standard deviation, first and third quartiles of the data. The proposed ADR architecture can yield an average percentage improvement of buffer fill rate up to 100.0 % with an average cost of 0.0058 and 0.0035 cents per second for routers and services, respectively. Regarding the mean of data over all overloading cases, i.e., those cases with $BFR(0,0) > 0$, the proposed architecture gives an enhancement of $\overline{\Delta BFR} = 46.7\%$ with $\overline{C} = 0.0024$ cents per second

Table 6.3.: Statistics of the Evaluation Data

	Evaluation Metric	***min***	***Q_1***	***median***	***Q_3***	***max***	***mean***	***σ***
Routers	$\overline{\Delta BFR}$ (%)	9.804	32.680	49.528	57.132	100.000	46.703	15.487
	$\overline{C}$	25e-5	15e-4	25e-4	32e-4	58e-4	24e-4	12e-4
Services	$\overline{\Delta BFR}$ (%)	29.412	52.632	60.475	69.754	100.000	61.848	15.654
	$\overline{C}$	5e-4	15e-4	16e-4	16e-4	35e-4	16e-4	6e-4

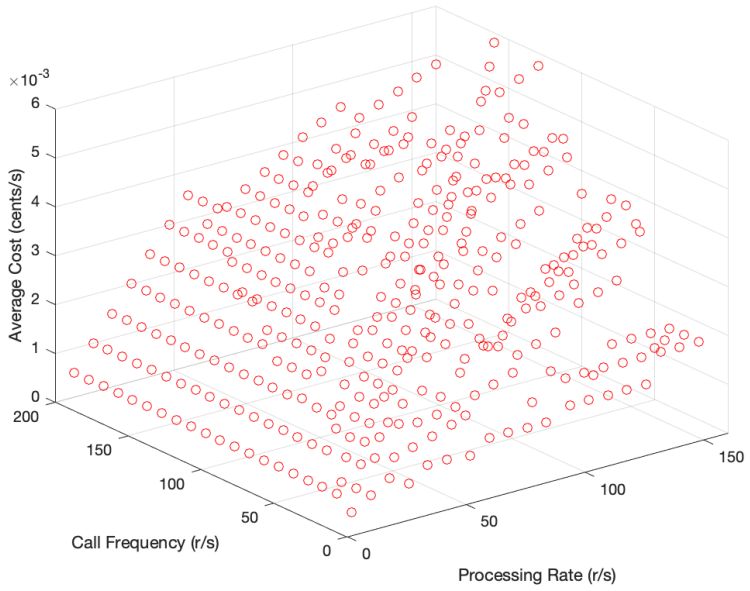


(a) $\overline{\Delta BFR}$ for Routers

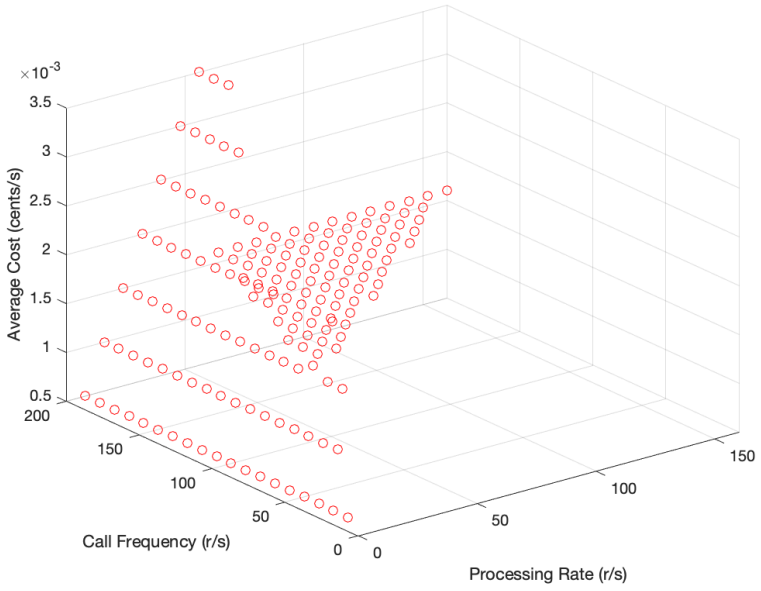


(b) $\overline{\Delta BFR}$ for Services

Figure 6.3.: Plots of Evaluation Data Regarding the Average Percentage Difference of Buffer Fill Rate



(a) $\overline{C}$ for Routers



(b) $\overline{C}$ for Services

Figure 6.4.: Plots of Evaluation Data Regarding the Average Reconfiguration Cost

for routers, and $\overline{\Delta BFR} = 61.8\%$ with $\overline{C} = 0.0016$ cents per second for services.

6.7. Threats to Validity

We study the threats to the validity of our study based on the four threat types by Wohlin et al. [96].

6.7.1. Construct Validity

The accurate representation of the intended construct by a measurement is assessed through construct validity. In this chapter, we studied reconfiguration measures of increasing the processing rate and scaling out a component to prevent system overload. While this is a common approach in service- and cloud-based research (see Section 1.3), the threat remains that other measures might work better in terms of system overload prevention, for instance, changing the routing technology by using a circuit breaker [78], or adding more routers and reconfiguring the routing.

6.7.2. Internal Validity

Internal validity concerns factors that affect the independent variables concerning causality. We considered each component, i.e., a router or a service, of a cloud-based application separately by modeling them as queuing stations. If a reconfiguration is required, our approach performs a multi-criteria optimization analysis [4] individually for the component. The threat remains that these components can have interdependencies, e.g., preventing overload of an upstream component might stress the downstream components. Moreover, we only considered constant load when modeling the stress of components using queuing theory. In reality, cloud-based systems are met with different load profiles, e.g., sudden load spikes because of a discount offer in an online shopping system. Those load profiles may require different models not covered in this study.

6.7.3. External Validity

External validity concerns threats that limit the ability to generalize the results beyond the experiment. We designed our novel architecture with generality in mind and explained how architects could specify the models to their needs. In spite of the fact that we systematically evaluated our approach with an extensive number of 9600 evaluation cases using the experiment infrastructure of our experiment of 1200 hours (see Section 2.3.2), the threat remains that evaluating the ADR architecture based on another infrastructure may lead to different results.

6.7.4. Conclusion Validity

Conclusion validity concerns factors that affect the ability to conclude the relations between treatments and study outcomes. As the statistical method to evaluate our model

in this chapter, we defined the average percentage difference of buffer fill rate $\overline{\Delta BFR}$ based on the Mean Absolute Percentage Error (MAPE) metric [90] as it is widely-used and offers good interpretability in our research context.

6.8. Conclusions

In this chapter, we extended our Adaptive Dynamic Routers (ADR) architecture. ADR uses queuing theory to model routers and services of a cloud-based system and performs multidimensional autoscaling on each component individually to prevent system overload. We evaluated our approach systematically using 9600 evaluation cases based on our empirical data[6]. Our results show that the proposed architecture yields up to 100% average percentage difference of buffer fill rate for routers and services. Regarding the mean of the data over cases where an overload is predicted, our approach reduces the average overload rate by 46.7% and 61.8% for routers and services, respectively. To the best of our knowledge, there has not been any cost-aware self-adaptive dynamic-routing architecture in the literature that adjusts the container configurations automatically to prevent system overload.

7. Stateful Container Depletion

This chapter studies the stateful depletion and rescheduling of the containers of idle components to address the research problem P_5: *Lack of an approach to statefully deplete and reschedule sporadically-active components*. In Chapter 8, we summarize the approach presented here and present a multifaceted reconfiguration of the dynamic-routing applications. We address the research problem P_6: *Lack of tool support for the multifaceted reconfiguration of dynamic routing applications*.

7.1. Introduction

Container scheduling is a fundamental part of today's service and cloud-based applications. Schedulers operate at different levels depending on how much control the system developers have. On the one hand, container orchestration managers such as Google Kubernetes[1] manage the scheduling of containers on different nodes. On the other hand, serverless managers, such as Google Autopilot[2], take care of the underlying infrastructure automatically and developers do not need to manage the nodes. However, regarding container depletion, i.e., removing the assigned cloud resources to an idle container, current scheduling technologies have limitations. In this chapter, we extend our Adaptive Dynamic Routers (ADR) architecture to manage cloud-resource usage efficiently when containers are idle. For this purpose, we deplete idle containers statefully, i.e., propose a novel manager that monitors idle containers, saves their state, and efficiently depletes them. This manager reconstructs a depleted container using the saved state when reconstruction is needed. In our approach, we suggest an Infrastructure as Code (IaC) component to automate the creation of new nodes if a depleted container cannot be scheduled on the same node, e.g., because of being overloaded. We provide an analytical model for the stateful depletion of containers and their rescheduling and empirically evaluate the accuracy of our model. For this purpose, we ran an experiment on a private cloud infrastructure and Google Cloud Platform[3] with a duration of 80 hours. Our model has a low error rate of 4.28% averaged over public and private clouds.

The structure of the chapter is as follows: In Section 7.2, we give the background of our study. Section 7.3 explains our approach details. Our analytical model is parameterized in Section 7.4 and evaluated in Section 7.5. In Section 7.6, we discuss the prediction error of our model and present the threats to the validity of our study. Finally, we conclude in Section 7.7.

[1] https://kubernetes.io
[2] https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview
[3] https://cloud.google.com

7.2. Background

This section gives the background of our study. Table 7.1 presents the mathematical notations used in this chapter.

Table 7.1.: The Mathematical Notations Used in this Chapter

Notation	Description
n_{busy}	Number of busy containers
n_{spor}	Number of sporadical containers
n_{extr}	Number of extra containers
n_{depl}	Number of depleted containers
cf_{busy}	Call frequency of busy containers
cf_{spor}	Call frequency of sporadical containers
cf_{extr}	Call frequency of extra containers
cf^c_{busy}	Call frequency of a busy container c
cf^c_{spor}	Call frequency of a sporadical container c
cf^c_{extr}	Call frequency of extra container c
T	Observed system time
T_{spor}	Active time of sporadical containers
T_{actv}	Overall active time of sporadical containers
T_{idle}	Idle time before depletion of sporadical containers
T_{extr}	Active time of extra containers
T^c_{spor}	Active time of a sporadical container c
T^c_{idle}	Idle time before depletion of a sporadical container c
T^c_{extr}	Time period an extra container c is active
d_{spor}	Delay of idleness of sporadical containers
d^c_{spor}	Delay of idleness of a sporadical container c
$\sum_{act}$	Summation of the periods a sporadical container c is active
r/s	Requests per second
PR	Number of processed requests without depletion
PR_{depl}	Number of processed requests with depletion
$MAPE$	Mean absolute percentage error
MAE	Mean absolute error
MSE	Mean squared error
MSE	Root mean squared error
ΔPR	Percentage improvement of the processed requests
ΔPR^c_{model}	Result of the model for an experiment case c
$\Delta PR^c_{empirical}$	Measured empirical data for an experiment case c
$Cases$	Set of the number of services and the number of routers
n_c	Length of $Cases$

7.2.1. Real-World Industrial Case Study

A case study with particular complexity is encountered by fiskaly GmbH, a provider of cloud-based Certified Technical Security Systems (CTSS) used to combat tax fraud. According to German legislation[4], every electronic cash register or Point of Sale (PoS) must be associated with a CTSS instance. This instance is responsible for tracking PoS business cases, recording them as event logs, and digitally signing and storing these logs for a future audit by the tax authorities. Each CTSS comprises two main components: A protocol unit, known as Security Module Application for Electronic Record-keeping Systems (SMAERS), and a cryptography unit, i.e., the cryptographic service provider. While multiple SMAERS instances can share the latter, each SMAERS must be assigned to a customer organization or, in practice, to a specific PoS of such an organization. In other words, the process data of each customer entity must be kept separate from all other customers.

As a result, at least one but potentially up to thousands of SMAERS instances, each running on its own container, must be managed for each customer. Consequently, hundreds of thousands of containers for the German market alone exist. These containers are clustered together in groups of more than 100 containers on virtual machines in the cloud. Depending on the customer's identity and the location of the associated PoS, a SMAERS instance may be highly active, processing thousands of transactions each day, or only sporadically active, with a handful of transactions per week. Larger organizations tend to batch-create SMAERS instances, meaning that some clusters may feature many high-load instances belonging to the same customer, while others have more mixed demography. Fiskaly GmbH needs to regulate resource consumption to ensure very high availability of the CTSS components per its service-level agreements and low-latency servicing of customer requests (typically under 250 ms). This regulation is necessary because cash-register transactions cannot be delayed for long. Furthermore, all signed logs generated by the CTSS have to be persisted and be constantly available for immediate export of auditing purposes by the tax authorities. This export includes formally decommissioned SMAERS instances, which must be kept active and accessible indefinitely.

7.2.2. Existing Solutions

Container Schedulers

Different business-grade container schedulers are available, e.g., Google Kubernetes[1] or Docker Swarm[5]. These tools usually work with constraints, with which a system designer controls on which cloud node a container is scheduled and deployed. However, these schedulers are mostly static: A designer must usually define the node pools and the constraints in advance. If a reconfiguration is needed, this information must be updated manually. Moreover, a container orchestration tool would provide more resources than needed when containers are idle, resulting in an increase in cloud costs. This increase is

[4] https://kassensichv.com
[5] https://docs.docker.com/engine/swarm/

because the depletion of containers cannot be done generically and must be tailored for each application separately.

Elastic Containers

Many cloud providers offer a standard service to automate this process, i.e., to run containers without considering the underlying cloud nodes. For instance, Google Autopilot[2], Microsoft Azure Container Instances[6] and Amazon Elastic Container Service[7] on Fargate[8], all of which free a designer from the container scheduling and deployment decisions. However, these solutions are not always applicable when a degree of control is needed over the underlying cloud nodes, e.g., following a compliance rule that dictates which servers store customer data. So the containers might violate deployment regulations in a fiscal application.

Controllers

Different container-scheduling technologies provide controllers that monitor the state of a cloud-based application. For example, using a controller in Kubernetes[9], the number of pods[10] can be changed automatically to achieve the desired state of an application. However, these controllers do not usually focus on a specific container but adjust a cluster of containers to achieve a goal for that cluster. Using a specific controller for each container requires additional effort. Furthermore, detailed knowledge of the scheduler is needed when extending a controller from a specific technology. Also, this process cannot be easily applied to another technology.

7.3. Approach Details

We study the scenario where containers can be depleted statefully, i.e., their state is saved for later reconstruction, and their resources are freed. When a new request to a depleted container is made, our proposed approach reconstructs the container. This container is scheduled on an existing node, or a new node is created. Our approach is generic and can be used with different container orchestration technologies, e.g., Google Kubernetes[1] and IaC tools such as Ansible[11]. We analyze the improvements in the number of processed requests when depleting idle containers statefully.

7.3.1. Definition of Container Types

Based on our studied industrial use case (see Section 7.2.1), we define *busy containers* as the ones that are active most of the time, processing requests with a high frequency,

[6] https://azure.microsoft.com/en-us/services/container-instances/
[7] https://aws.amazon.com/ecs/
[8] https://aws.amazon.com/fargate/
[9] https://kubernetes.io/docs/concepts/architecture/controller/
[10] https://cloud.google.com/kubernetes-engine/docs/concepts/pod
[11] https://www.ansible.com

i.e., thousands of requests per day. On the other hand, some containers receive requests with a lower frequency and are idle between incoming calls. In this chapter, we call these sporadically-active containers *sporadical containers*. In our approach, we deplete these containers when idle and deploy *extra containers* instead to improve resource usage. Let n_{depl} be the number of depleted containers and n_{extr} the number of extra containers.

7.3.2. Reconfiguration Algorithms

Algorithm 3 proposes a simple depletion strategy that is used by the *Manager* component in Figure 2.5. We monitor sporadical containers for idleness. When depletion is triggered for a container, we replace it with an extra container, i.e., a new container that has yet to be deployed. However, deploying an extra container only happens if there are more than twice as many depleted containers as extra containers in the system. This policy ensures we do not overload the cloud nodes by deploying extra containers after each depletion.

Algorithm 3: Reconfiguration Algorithm to Statefully Deplete and Deploy Extra Containers

Input: n_{depl}, n_{extr}

foreach *container : containers* **do**
  if *isActive(container) == false* **then**
    deplete(*container*)
    $n_{depl} \leftarrow n_{depl} + 1$
    if $n_{depl} >= 2 \cdot n_{extr} + 1$ **then**
      replace_extra_containers()
      $n_{extr} \leftarrow n_{extr} + 1$
    end
  end
end
return n_{extr}

Algorithm 4 shows a simple scheduling algorithm when a request for a depleted container arrives. In this case, the container is scheduled on an existing node. Alternatively, if the current nodes are overloaded, the IaC component creates a new node on which the container is scheduled. Section 7.6 presents an analysis to recognize and predict an overloaded node. Moreover, in Chapter 8, we introduce more details of the IaC algorithm by considering the capacity of nodes, which is relevant when studying a multifaceted reconfiguration of dynamic routing. As a result, these details are not introduced here. Note that the number of depleted containers n_{depl} affects the number of extra containers n_{extr} used in our analytical model.

7.3.3. Analytical Model

We present the analytical model of our approach in this chapter.

Algorithm 4: Reconfiguration Algorithm to Schedule a Depleted Container

```
Input: n_depl
foreach container : depleted_containers do
    if isActive(container) == true then
        if nodes_overloaded() == false then
            deploy_on_current_nodes(container)
        else
            deploy_on_new_node(container)
        end
        n_depl ← n_depl − 1
    end
end
return n_depl
```

Definition of Depletion Events

To calculate the achieved improvements in terms of the number of processed requests when depleting idle containers statefully, we need a base measurement to compare with. That is the number of requests that are processed without any depletion of containers. For this purpose, our approach uses a combination of busy and sporadical services. We calculate the number of requests processed during a fixed observed time and analytically model this measurement of requests without depletion. Moreover, we model the depletion of sporadical containers when idle, and calculate the improvements in the number of processed requests when busy containers are scheduled instead.

Load Profiles

Let T be the observed system time in seconds, n_{busy} the number of busy containers in a system, and cf^c_{busy} the call frequency of a busy container c. We define a busy load profile as constantly feeding a busy container c with a call frequency of cf^c_{busy} without stopping during T.

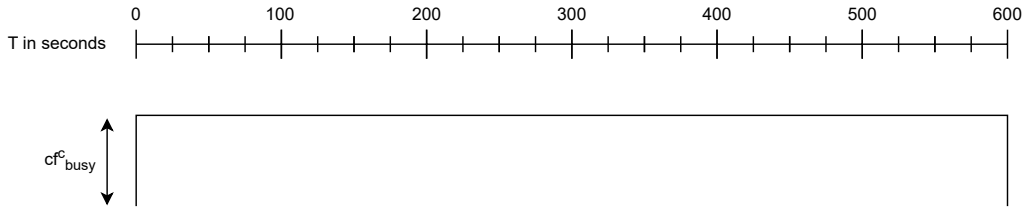


Figure 7.1.: The Load Profile of a Busy Container c

Let n_{spor} be the number of sporadical containers in an application, and cf^c_{spor}, d^c_{spor}, T^c_{spor}, and T^c_{idle} be the call frequency, the delay of idleness, the active time, and the idle

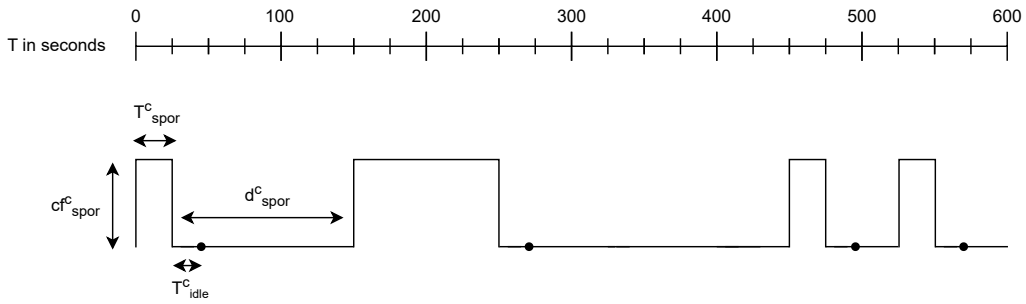


Figure 7.2.: The Load Profile of a Sporadical Container c (dots represent depletion.)

time before depletion for a sporadical container c. As shown in Figure 7.2, a sporadical load profile for a container c is characterized by a call frequency of cf^c_{spor} for a short time T^c_{spor} followed by a delay d_{spor} of no incoming requests. Note that a sporadical load is repeated with different values of time and delay.

Number of Requests without Depletion Events

Let PR be the number of processed requests without depletion and $\sum_{act}$ the summation of the periods a sporadical container c is active. PR can be calculated as the total number of requests processed by the busy containers plus the requests processed by the sporadical containers when active:

$$PR = \sum_{c=1}^{n_{busy}} cf^c_{busy} \cdot T + \sum_{c=1}^{n_{spor}} \sum_{act} cf^c_{spor} \cdot T^c_{spor} \tag{7.1}$$

Number of Requests with Depletion Events

We deplete a sporadical container c when it is idle for a period of time T^c_{idle} (represented by dots in Figure 7.2). We deploy extra containers instead according to Algorithm 3. Let cf^c_{extr} be the call frequency of extra container c, T^c_{extr} the time period an extra container c is active, and PR_{depl} the number of processed requests with depletion:

$$PR_{depl} = PR + \sum_{c=1}^{n_{extr}} \sum_{act} cf^c_{extr} \cdot T^c_{extr} \tag{7.2}$$

Let ΔPR be the percentage improvement of the processed requests. Using Equation (7.2), we have:

$$\Delta PR = \frac{100\%}{PR} \cdot (PR_{depl} - PR) \tag{7.3}$$

$$\Delta PR = \frac{100\%}{PR} \cdot \sum_{c=1}^{n_{extr}} \sum_{act} cf^c_{extr} \cdot T^c_{extr} \tag{7.4}$$

7.4. Parameterization of Model Elements

Our analytical model can be applied to different scenarios of multiple load profiles with various numbers of containers. Architects must parameterize our ADR model to their specific use case at hand. In this section, we introduce an illustrative sample case and explain how this parameterization of our model elements can be performed.

7.4.1. Illustrative Sample Case

We consider a scenario where the load profiles of multiple sporadical containers are so that these containers can be swapped without losing any requests. In this case, when a container is idle, another container receives incoming calls, as shown in Figure 7.3 for an example. The main benefit of this case is that the node's resources, e.g., vCPUs, are not reserved for an idle container and can be used efficiently for a busy container, resulting in faster response time. This efficient usage can also reduce costs depending on the cloud-cost profile a user opts for. If customers are billed per resource usage (e.g., see Google Autopilot pricing[12]), the cost reduction can be significant.

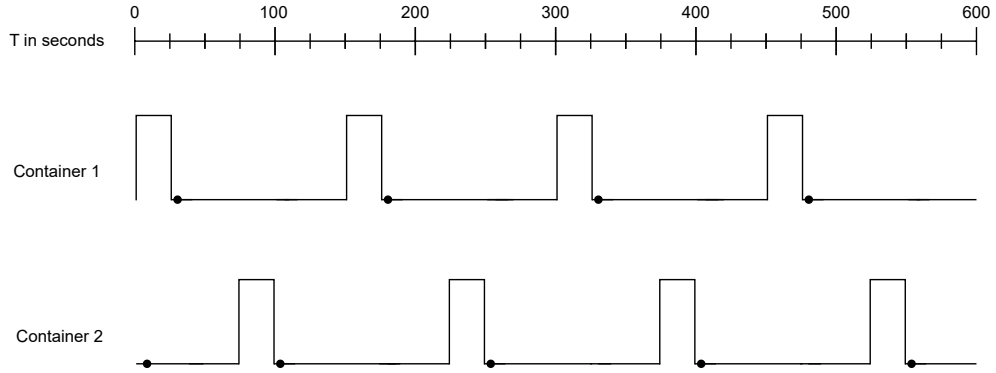


Figure 7.3.: The Sporadical Load Profiles of Two Containers in the Illustrative Sample Case (dots represent depletion.)

7.4.2. Model Parameterization for the Sample Case

In Figure 7.3, the sporadical load profile is repeated homogeneously with constant values. Let cf_{busy} be the call frequency of busy containers, and cf_{spor}, T_{spor}, and d_{spor} the call frequency, active time, and delay of idleness of sporadical containers. As a result of the homogeneous load for sporadical containers, we can calculate the overall active time of the sporadical containers as a fraction of T. Let T_{actv} be the overall active time of sporadical

[12] https://cloud.google.com/kubernetes-engine/pricing

containers, we have:

$$T_{actv} = T \cdot \frac{T_{spor}}{T_{spor} + d_{spor}} \tag{7.5}$$

We can calculate the total requests processed by busy and sporadical containers by rewriting Equation (7.1) for the homogeneous load as:

$$PR = n_{busy} \cdot cf_{busy} \cdot T + n_{spor} \cdot cf_{spor} \cdot T \cdot \frac{T_{spor}}{T_{spor} + d_{spor}} \tag{7.6}$$

To clarify, we study constant values for call frequencies, active times, and delays of all containers in our sample case. Hence, Equation (7.1) can be rewritten as Equation (7.6).

Let cf_{extr} and T_{extr} be the call frequency and active time of extra containers, respectively. We can rewrite Equation (7.2) for the illustrative sample case as:

$$PR_{depl} = PR + n_{extr} \cdot cf_{extr} \cdot T_{extr} \tag{7.7}$$

Following Algorithm 3, the n_{extr} is half the number of sporadical containers in the sample case. The reasoning behind this is that two containers are swapped repeatedly, as it can be seen in Figure 7.3, and one container is replaced in this case:

$$n_{extr} = \frac{n_{spor}}{2} \tag{7.8}$$

We mentioned that when we deplete idle containers, we replace them with busy containers, i.e., we feed them with a busy load profile to maximize the number of processed requests, therefore:

$$cf_{extr} = cf_{busy} \tag{7.9}$$

Let T_{idle} be the idle time after which we deplete sporadical containers. In this illustrative sample case, the extra containers are deployed after the first depletion happens, i.e., after T_{idle}, and are active for the rest of the experiment:

$$T_{extr} = T - T_{idle} \tag{7.10}$$

Finally, the percentage improvement of the processed requests, i.e., ΔPR, presented in Equation (7.4) for this scenario is:

$$\Delta PR = \frac{100\%}{PR} \cdot n_{extr} \cdot cf_{extr} \cdot T_{extr} \tag{7.11}$$

$$\Delta PR = \frac{100\%}{PR} \cdot \frac{n_{spor}}{2} \cdot cf_{busy} \cdot (T - T_{idle}) \tag{7.12}$$

7.5. Evaluation

To evaluate our approach, we designed an experiment on cloud settings representative of our industrial case study (see Section 7.2.1).

7.5.1. Experiment Planning

In this chapter, we introduce an experiment with a runtime duration of 80 hours.

Goal

We aim to empirically evaluate the improvement in efficient resource usage when idle containers are depleted statefully and reconstructed at a later time.

Method

We containerize the number of services representative of our industrial case study and deploy these containers on a virtual node in public and private cloud infrastructures (see below for details). Afterward, requests with different levels of frequencies are sent to these containerized services. We deplete idle containers for a period of time and measure the difference in the number of processed requests with and without depletion. Our experiment planning follows the illustrative sample case presented in Section 7.4.1. Moreover, we use private and public cloud infrastructures to validate the accuracy of our model.

Private Cloud Infrastructure

We used a physical server having two identical CPUs, the Intel® Xeon® E5-2680 v4 @ 2.40 GHz[13]. Each processor has 14 cores and two physical threads per core (56 in total). We installed a virtual node on the server using VMware ESXi version 6.7.0 u2 hypervisor. This virtual node has eight vCPU cores, 60 GB system memory, and runs Ubuntu Server 18.04.01 LTS[14]. Docker technology[15] is used to containerize services implemented in Node.js[16]. Each service listens for a request and performs a dummy operation, i.e., a delay of 1000 loops.

Validation Experiment on Public and Private Clouds

We used our private cloud to have control over the infrastructure. On a public cloud, other factors, such as the parallel workload of other applications, can influence the results. To show that our approach can be used on other infrastructures as well, we empirically validate the analysis of our proposed model on our private cloud infrastructure and Google Cloud Platform (GCP)[3]. On GCP, we use two machine types, i.e., general-purpose E2 machine instance[17] with two vCPUs and 8 GB of memory, and compute-optimized C2 machine instance[18] with four vCPUs and 16 GB of memory. We duplicated our private

[13] https://www.intel.com/content/www/us/en/homepage.html
[14] https://www.ubuntu.com
[15] https://www.docker.com
[16] https://nodejs.org/en/
[17] https://cloud.google.com/compute/docs/general-purpose-machines
[18] https://cloud.google.com/compute/docs/compute-optimized-machines

cloud infrastructure on these machines and repeated the experiment. Overall we have three repetitions that we call: **Private**, **GCP2** (two vCPUs), and **GCP4** (four vCPUs).

Load Generation

For load generation, we utilized a MacBook Pro with an Apple M1 Pro chip and 16 GB of system memory that runs macOS Monterey[19] version 12.2.1. It generates load using Apache JMeter[20] that sends hypertext transfer protocol version 1.1[21] requests to the virtual nodes.

Experiment Cases

Like our other experiments (see Section 2.3.2), we take the duration, i.e., the observed time, of $T = 600$ seconds. We define two load profiles, i.e., *busy* and *sporadical profiles* based on our industrial case study (see Section 7.2.1). A busy load profile is active during the entire experiment run. In our industrial case, busy containers can process thousands of requests daily. Therefore, we define a representative call frequency of busy containers based on requests per second (r/s):

$$cf_{busy} = 5\ r/s \tag{7.13}$$

Note that these frequencies result in 432000 requests per day.

Presenting our case study, we mentioned that a sporadical load could be as low as a handful of weekly requests. As this is not predictive and might result in no requests per $T = 600$ seconds of experiment time, we follow the sporadical load profiles presented in Figure 7.3. To observe some requests during our experiment time (and for the cases to be comparable), we give the same call frequency for sporadical and extra containers as for the busy containers (i.e., $5\ r/s$):

$$cf_{spor} = cf_{extr} = 5\ r/s \tag{7.14}$$

However, as mentioned before, the sporadical load is active for a time period T_{spor} and inactive for a short delay d_{spor}. To more closely resemble our industrial case study and cover multiple sporadical load profiles, we use two levels for T_{spor} and d_{spor}:

$$(T_{spor}, d_{spor}) \in \{\ (25, 125)\,, (50, 150)\ \} \tag{7.15}$$

Containers are depleted after a T_{idle} of inactivity. To study the effects of this model element, we also use three levels as the following:

$$T_{idle} \in \{5, 25, 45\}\ \ s \tag{7.16}$$

[19] https://www.apple.com/by/macos/monterey/
[20] https://jmeter.apache.org
[21] https://tools.ietf.org/html/rfc7230

These levels are chosen so we have a very short delay of 5 s and the longest delay of 45 s that still results in a homogeneous load, as used in our illustrative sample case. Moreover, we study a middle case, i.e., a delay of 25 s to be thorough.

Following our industrial use case, we use different numbers of busy and sporadical containers. Remember that the number of extra containers in the illustrative sample case is half the number of sporadical containers according to Equation (7.8) (see section Section 7.4.2 for explanation):

$$\begin{aligned}(n_{busy}, n_{spor}, n_{extr}) \in \{\ & (50, 50, 25), \\ & (80, 20, 10), \\ & (100, 20, 10), \\ & (100, 50, 25)\ \}\end{aligned} \tag{7.17}$$

We have 96 evaluation cases in total. These include 32 experiment cases (with and without depletion) validated on three cloud infrastructures, i.e., private, GCP2, and GCP4. Moreover, each case was repeated five times and averaged over to mitigate the influence of other factors on the measurements, such as other workloads in public clouds (see threats to validity Section 7.6.3). Overall, our experiment had a run-time of 80 hours, excluding setup and processing time. Details of each evaluation case are provided in the online artifact of this thesis[6].

7.5.2. Experiment Results

Table 7.2 presents our model predictions and empirical measurements, and Figure 7.4 visualizes the data. We analyze the results separately for private and public cloud infrastructures.

Private Cloud Infrastructure

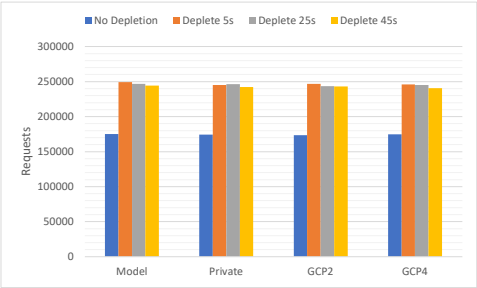
The experiment results are very close to our analytical model predictions in all cases of our private cloud infrastructure. Our model predicts, and our experiment confirms that when having the same number of containers and feeding them with the same load profile, a shorter T_{idle} (the time period a container is idle before depletion) results in a higher PR_{depl} (number of processed requests with depletion). This improvement is because we quickly identify idle containers and replace them with busy ones. Therefore, more requests are processed during the same period of time, i.e., resources are used more efficiently.

Moreover, when sporadical load profiles have a longer T_{spor} (the time period a sporadical container is active), we have a higher PR_{depl} in all cases when we keep other model elements constant. This improvement is because when sporadical containers are swapped, they stay active longer, processing more requests. Another interesting observation is that the ratio of busy to sporadical containers directly impacts ΔPR, i.e., the percentage improvement of the processed requests. In all experiment cases with $(n_{busy}, n_{spor}, n_{extr})$ of $(80, 20, 10)$ and $(100, 20, 10)$, we have around 10% increase in the number of the processed

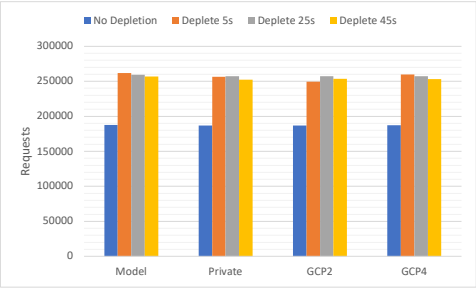
Table 7.2.: Model Predictions of Experiment Cases and Empirical Results

$T_{idle}(s)$	**Num. of Containers** $(n_{busy}, n_{spor}, n_{extr})$	**Load Profile** (T_{spor}, d_{spor})	PR	PR_{depl}	$\Delta PR(\%)$	PR	PR_{depl}	$\Delta PR(\%)$
			Model			**Private**		
5	(50, 50, 25)	(25, 125)	175000.00	249375.00	42.50	174180.00	245020.00	40.67
		(50, 150)	187500.00	261875.00	39.87	186810.00	256480.00	37.29
	(80, 20, 10)	(25, 125)	250000.00	279750.00	11.90	250756.00	280348.00	11.80
		(50, 150)	255000.00	284750.00	11.67	255580.00	285244.00	11.61
	(100, 20, 10)	(25, 125)	310000.00	339750.00	9.60	310988.00	340580.00	9.51
		(50, 150)	315000.00	344750.00	9.44	315780.00	345462.00	9.40
	(100, 50, 25)	(25, 125)	325000.00	399375.00	22.88	322420.00	388299.00	20.43
		(50, 150)	337500.00	411875.00	22.04	335210.00	404290.00	20.61
25	(50, 50, 25)	(25, 125)	175000.00	246875.00	41.07	174180.00	246335.00	41.42
		(50, 150)	187500.00	259375.00	38.33	186810.00	257460.00	37.82
	(80, 20, 10)	(25, 125)	250000.00	278750.00	11.50	250756.00	279234.00	11.36
		(50, 150)	255000.00	283750.00	11.27	255580.00	284204.00	11.20
	(100, 20, 10)	(25, 125)	310000.00	338750.00	9.27	310988.00	339314.00	9.11
		(50, 150)	315000.00	343750.00	9.13	315780.00	344224.00	9.01
	(100, 50, 25)	(25, 125)	325000.00	396875.00	22.12	322420.00	392450.00	21.72
		(50, 150)	337500.00	409375.00	21.30	335210.00	402045.00	19.94
45	(50, 50, 25)	(25, 125)	175000.00	244375.00	39.64	174180.00	242355.00	39.14
		(50, 150)	187500.00	256875.00	37.00	186810.00	252295.00	35.05
	(80, 20, 10)	(25, 125)	250000.00	277750.00	11.10	250756.00	278364.00	11.01
		(50, 150)	255000.00	282750.00	10.88	255580.00	283048.00	10.75
	(100, 20, 10)	(25, 125)	310000.00	337750.00	8.95	310988.00	338080.00	8.71
		(50, 150)	315000.00	342750.00	8.81	315780.00	343366.00	8.74
	(100, 50, 25)	(25, 125)	325000.00	394375.00	21.35	322420.00	387565.00	20.20
		(50, 150)	337500.00	406875.00	20.55	335210.00	401460.00	19.76

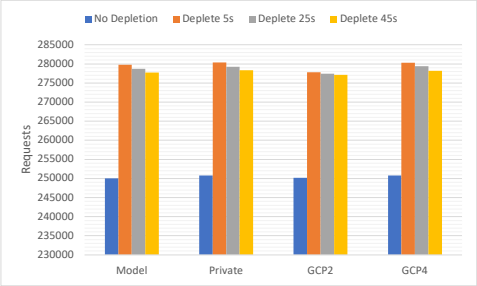
			GCP2			**GCP4**		
5	(50, 50, 25)	(25, 125)	173330.00	246935.00	42.47	174650.00	245995.00	40.85
		(50, 150)	187060.00	249585.00	33.43	187260.00	259580.00	38.62
	(80, 20, 10)	(25, 125)	250200.00	277796.00	11.03	250764.00	280276.00	11.77
		(50, 150)	254924.00	283392.00	11.17	257532.00	285234.00	10.76
	(100, 20, 10)	(25, 125)	308048.00	333166.00	8.15	310904.00	340314.00	9.46
		(50, 150)	309600.00	335316.00	8.31	318492.00	345512.00	8.48
	(100, 50, 25)	(25, 125)	320720.00	374910.00	16.90	324540.00	393010.00	22.00
		(50, 150)	325050.00	370295.00	13.92	336530.00	407845.00	21.19
25	(50, 50, 25)	(25, 125)	173330.00	243690.00	40.59	174650.00	245295.00	40.44
		(50, 150)	187060.00	257365.00	37.58	187260.00	257140.00	37.32
	(80, 20, 10)	(25, 125)	250200.00	277412.00	10.88	250764.00	279374.00	11.41
		(50, 150)	254924.00	281530.00	10.44	257532.00	284286.00	10.39
	(100, 20, 10)	(25, 125)	308048.00	332460.00	7.92	310904.00	339402.00	9.17
		(50, 150)	309600.00	334226.00	7.95	318492.00	344340.00	8.12
	(100, 50, 25)	(25, 125)	320720.00	369435.00	15.19	324540.00	395080.00	21.74
		(50, 150)	325050.00	372150.00	14.49	336530.00	404780.00	20.28
45	(50, 50, 25)	(25, 125)	173330.00	243240.00	40.33	174650.00	240820.00	37.89
		(50, 150)	187060.00	253500.00	35.52	187260.00	253210.00	35.22
	(80, 20, 10)	(25, 125)	250200.00	277128.00	10.76	250764.00	278156.00	10.92
		(50, 150)	254924.00	280468.00	10.02	257532.00	282176.00	9.57
	(100, 20, 10)	(25, 125)	308048.00	329408.00	6.93	310904.00	338094.00	8.75
		(50, 150)	309600.00	333302.00	7.66	318492.00	343506.00	7.85
	(100, 50, 25)	(25, 125)	320720.00	362720.00	13.10	324540.00	392650.00	20.99
		(50, 150)	325050.00	364810.00	12.23	336530.00	398930.00	18.54



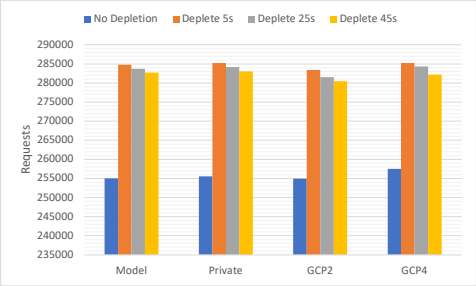
(a) Num. of Containers: $(50, 50, 25)$
Load Profile: $(25, 125)$



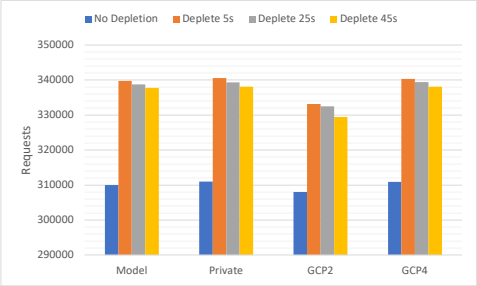
(b) Num. of Containers: $(50, 50, 25)$
Load Profile: $(50, 150)$



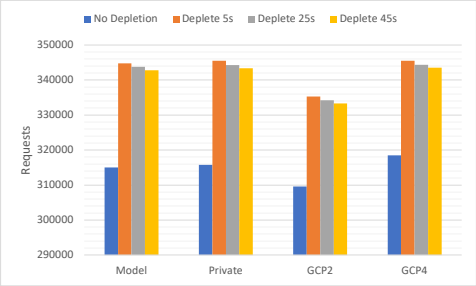
(c) Num. of Containers: $(80, 20, 10)$
Load Profile: $(25, 125)$



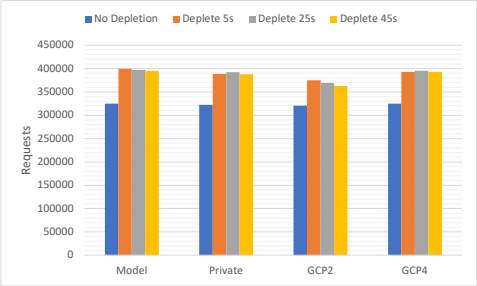
(d) Num. of Containers: $(80, 20, 10)$
Load Profile: $(50, 150)$



(e) Num. of Containers: $(100, 20, 10)$
Load Profile: $(25, 125)$



(f) Num. of Containers: $(100, 20, 10)$
Load Profile: $(50, 150)$



(g) Num. of Containers: $(100, 50, 25)$
Load Profile: $(25, 125)$

(h) Num. of Containers: $(100, 50, 25)$
Load Profile: $(50, 150)$

Figure 7.4.: Plots of All Cases without Depletion, and Depletion with T_{idle} seconds

requests. As we increase the number of sporadical containers (and consequently the number of extra containers) in the experiment case of $(100, 50, 25)$, the ΔPR also rises to a value close to 20%. A one-to-one ratio of busy and sporadical containers results in the highest ΔPR as predicted by our models and confirmed by our experiment of $(50, 50, 25)$ containers.

Public Cloud Infrastructure

The experiment measurements on GCP follow the data trend of the private cloud. However, as seen in Figure 7.4, as the number of containers goes higher, GCP2 has lower processed requests compared to our model predictions and other experiment infrastructures. This deterioration is because GCP2 has an E2 machine instance with two vCPUs. This machine can handle a lower number of containers, i.e., $(n_{busy}, n_{spor}, n_{extr})$ of $(50, 50, 25)$ and $(80, 20, 10)$, closely to the other experiment infrastructures. Nonetheless, when a high number of containers are deployed on this machine, i.e., $(100, 20, 10)$ and $(100, 50, 25)$ containers, the E2 machine is saturated and results in fewer processed requests. As seen in Table 7.2 and Figure 7.4, a more powerful C2 machine in GCP4 with four vCPUs can handle all our experiment cases, and the experiment results are close to our model predictions.

7.6. Discussion

This section studies container migration. Moreover, we calculate the prediction accuracy of our model and present the threats to the validity of our study.

7.6.1. Container Migration

As we studied in Section 7.5.2, the GCP2 infrastructure becomes saturated when the number of containers increases. The IaC component can automatically start a new machine to migrate the depleted containers, i.e., schedule the containers on a newly-created machine (see Figure 2.5). In our experiment, the GCP2 infrastructure performed close to our model predictions with $(n_{busy}, n_{spor}, n_{extr})$ of $(50, 50, 25)$ and $(80, 20, 10)$. That is when there are up to 100 deployed containers in a system, i.e., busy and sporadical containers.

$$n_{busy} + n_{spor} \leq 100 \tag{7.18}$$

However, with more than 100 deployed containers, i.e., $(n_{busy}, n_{spor}, n_{extr})$ of $(100, 20, 10)$ and $(100, 50, 25)$, the GCP4 infrastructure gave values close to our model predictions. Therefore, we can empirically conclude that for our experiment cases, the decision point to start a new machine is when we have 100 deployed containers. We call this **GCP Mixed**. The IaC component uses the following formula to change the infrastructure and

schedules a container using Algorithm 4.

$$\text{GCP Mixed} = \begin{cases} \text{GCP2} & \text{if } (n_{busy} + n_{spor} \leq 100) \\ \text{GCP4} & \text{otherwise} \end{cases}$$

Table 7.3 presents the model predictions of our experiment cases and the empirical measurements on this infrastructure.

7.6.2. Evaluation of the Prediction Error

We measure the accuracy of our model predictions compared to the empirical results of our private infrastructure (see Table 7.2) and the GCP Mixed infrastructure (see Table 7.3). We calculate the prediction error by calculating four error measurements commonly used in cloud research, i.e., Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE) [90]. We calculate the error measurements in terms of ΔPR. Let ΔPR^c_{model} and $\Delta PR^c_{empirical}$ be the result of the model, and the measured empirical data for an experiment case c, and n_c be the number of measured empirical cases. We have $n_c = 24$ ΔR values per each experiment infrastructure (see Tables 7.2 and 7.3). The error measurements are

Table 7.3.: Model Predictions and Empirical Results on the GCP Mixed Infrastructure

$T_{idle}(s)$	Num. of Containers $(n_{busy}, n_{spor}, n_{extr})$	Load Profile (T_{spor}, d_{spor})	PR	PR_{depl}	$\Delta PR(\%)$	PR	PR_{depl}	$\Delta PR(\%)$
			Model			**GCP Mixed**		
5	(50, 50, 25)	(25, 125)	175000.00	249375.00	42.50	173330.00	246935.00	42.47
		(50, 150)	187500.00	261875.00	39.87	187060.00	249585.00	33.43
	(80, 20, 10)	(25, 125)	250000.00	279750.00	11.90	250200.00	277796.00	11.03
		(50, 150)	255000.00	284750.00	11.67	254924.00	283392.00	11.17
	(100, 20, 10)	(25, 125)	310000.00	339750.00	9.60	310904.00	340314.00	9.46
		(50, 150)	315000.00	344750.00	9.44	318492.00	345512.00	8.48
	(100, 50, 25)	(25, 125)	325000.00	399375.00	22.88	324540.00	393010.00	22.00
		(50, 150)	337500.00	411875.00	22.04	336530.00	407845.00	21.19
25	(50, 50, 25)	(25, 125)	175000.00	246875.00	41.07	173330.00	243690.00	40.59
		(50, 150)	187500.00	259375.00	38.33	187060.00	257365.00	37.58
	(80, 20, 10)	(25, 125)	250000.00	278750.00	11.50	250200.00	277412.00	10.88
		(50, 150)	255000.00	283750.00	11.27	254924.00	281530.00	10.44
	(100, 20, 10)	(25, 125)	310000.00	338750.00	9.27	310904.00	339402.00	9.17
		(50, 150)	315000.00	343750.00	9.13	318492.00	344340.00	8.12
	(100, 50, 25)	(25, 125)	325000.00	396875.00	22.12	324540.00	395080.00	21.74
		(50, 150)	337500.00	409375.00	21.30	336530.00	404780.00	20.28
45	(50, 50, 25)	(25, 125)	175000.00	244375.00	39.64	173330.00	243240.00	40.33
		(50, 150)	187500.00	256875.00	37.00	187060.00	253500.00	35.52
	(80, 20, 10)	(25, 125)	250000.00	277750.00	11.10	250200.00	277128.00	10.76
		(50, 150)	255000.00	282750.00	10.88	254924.00	280468.00	10.02
	(100, 20, 10)	(25, 125)	310000.00	337750.00	8.95	310904.00	338094.00	8.75
		(50, 150)	315000.00	342750.00	8.81	318492.00	343506.00	7.85
	(100, 50, 25)	(25, 125)	325000.00	394375.00	21.35	324540.00	392650.00	20.99
		(50, 150)	337500.00	406875.00	20.55	336530.00	398930.00	18.54

calculated as follows:

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c=1}^{n_{case}} \left| \frac{\Delta PR^c_{model} - \Delta PR^c_{empirical}}{\Delta PR^c_{empirical}} \right| \tag{7.19}$$

$$MAE = \frac{1}{n_c} \cdot \sum_{c=1}^{n_{case}} \left| \Delta PR^c_{model} - \Delta PR^c_{empirical} \right| \tag{7.20}$$

$$MSE = \frac{1}{n_c} \cdot \sum_{c=1}^{n_{case}} \left(\Delta PR^c_{model} - \Delta PR^c_{empirical} \right)^2 \tag{7.21}$$

$$RMSE = \sqrt{MSE} \tag{7.22}$$

Table 7.4 Presents the prediction error of the proposed model compared to the measured data on different experiment infrastructures. Our model has a MAPE prediction error of 4.28% averaged over Private and GCP infrastructures. Given the 30.0% target prediction accuracy commonly used in the cloud quality-of-service field [59], The prediction error of our approach is more than reasonable. Other low error measurements also confirm the high accuracy of our model.

Table 7.4.: Prediction Error of the Proposed Model

Measurement	Private	GCP Mixed	Overall
$MAPE(\%)$	2.96	5.60	4.28
MAE	0.69	0.95	0.82
MSE	1.11	2.40	1.33
$RSME$	1.05	1.55	1.3

7.6.3. Threats to Validity

As in all empirical research, there are several threats to the validity and limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [96].

Construct Validity

The accurate representation of the intended construct by a measurement is assessed through construct validity. In our study, we modeled the depletion of containers based on the number of requests they process at a given time period. This approach is a common criterion in the cloud quality-of-service research (see state of the art in Section 1.3) and in current container scheduling technologies such as Google Kubernetes[1] to define controllers based on the incoming load. However, a threat remains that other measures, such as CPU usage percentage, might result in more accurate modeling of container depletion. More research with several real-world systems would be needed to cover other measurements and exclude this threat.

Internal Validity

Internal validity concerns factors that affect the independent variables concerning causality. We performed an experiment based on our studied industrial use case to evaluate our proposed model. However, we did so in limited experiment time and had control over the workload on cloud infrastructures. We avoided factors such as other loads on the machines where the experiment ran. To mitigate this threat, we repeated each experiment case five times and averaged the empirical measurements. However, more research with multiple real-world workloads would be needed to confirm that no other factors influence the measurements.

External Validity

External validity concerns threats that limit the ability to generalize the results beyond the experiment. We designed our approach with generality in mind and explained in detail how architects could tailor ADR to their needs (see Section 7.4). Although we evaluated our approach by designing a representative experiment and measuring empirical data, the threat remains that evaluating based on another infrastructure may lead to different results. To mitigate this thread, we validated our measurements on Google Cloud Platform infrastructure and showed that our results are applicable (see Section 7.5). Also, we considered multiple load profiles, including a sporadical load profile (see Section 7.3.3 and Figure 7.2). However, the load was constant regarding the frequency of incoming calls during active periods. A related threat is that we implemented all our services with Node.js[16] and did not use an off-the-shelf implementation, e.g., Envoy[22]. We did so to have a comparable infrastructure and to avoid technological impacts on our results.

Conclusion Validity

Conclusion validity concerns factors that affect the ability to draw conclusions about the relations between treatments and study outcomes. As the statistical method to compare our model's predictions to the empirical data, we used the MAPE metric as it is widely used and offers good interpretability in our research context. To mitigate the threat that this statistical method might have issues, we double-checked three other error measures, i.e., MAE, MSE, and RMSE, which confirmed our results reported in Section 7.6.2.

7.7. Conclusions

In this chapter, we proposed the details of our Adaptive Dynamic Routers (ADR) architecture. Our approach is based on MAPE-K [47, 16, 17] loops to monitor containers for idleness and deplete them if necessary. For this purpose, we proposed an analytical model. We explained the details of our approach that can also work with off-the-shelf orchestration solutions adding depletion-management capabilities. Moreover, we discussed how architects could parameterize our analytical model to different scenarios by following

[22] https://www.envoyproxy.io/

an illustrative sample case. For the empirical validation of our models, we designed and performed an experiment on a private cloud infrastructure, as well as on Google Cloud Platform[3]. Based on the details of our studied industrial use case (see Section 7.2.1), we defined multiple experiment cases and compared our empirical results to our model predictions.

We found out empirically that, for our experiment, 100 deployed containers is a decision point to start a new machine automatically using the IaC component of our proposed approach (see Figure 2.5). We calculated the prediction error of our model as 4.28% based on the widely-used mean absolute percentage error [90] averaged over private and public clouds. To the best of our knowledge, stateful depletion of containers has yet to be extensively studied in the literature. Moreover, current container orchestration technologies, such as Google Kubernetes[1], consider container depletion minimally. We believe our proposed approach can provide a solid base for further research in this area.

8. Multifaceted Reconfiguration

This chapter studies the interconnection between different configuration views studied in Chapters 5 to 7. We introduce a multifaceted reconfiguration of dynamic routing to address the research problem P_6: *Lack of tool support for the multifaceted reconfiguration of dynamic routing applications*. We investigate three scenarios, i.e., when components are idle, active, and overloaded. Moreover, this chapter provides prototypical tool support to demonstrate our reconfiguration concepts.

8.1. Introduction

Dynamic reconfiguration is commonly used in service- and cloud-based applications. In combination with autoscalers, dynamic routers can adapt the system to the resource demands, e.g., in an e-commerce application offering discounts for services in a specific location. Without such measures, the quality-of-service metrics are affected negatively, and a system overload can lead to an application being non-responsive. However, our Adaptive Dynamic Routers (ADR) architecture must consider the cost of cloud resource usage when performing these reconfiguration steps to avoid adding high additional costs. This chapter proposes a cost-aware multifaceted reconfiguration of dynamic routing applications. We study the depletion and rescheduling of idle containers and use an Infrastructure as Code (IaC) component to apply changes to the infrastructure. Moreover, when system components, i.e., the routers and services, are in a steady state, our approach dynamically self-adapts between more central or distributed routing to optimize reliability and performance. This adaptation is calculated based on a system-wide optimization analysis. When system components are overloaded, we perform a per-component optimization to autoscale the system multidimensionally. Our extensive systematic evaluation shows improvements in quality trade-off adaptations and system overload prevention. We provide prototypical tool support to demonstrate our concepts with illustrative sample cases.

The structure of the chapter is as follows: Section 8.2 presents an approach overview. Section 8.3 explains our approach in detail, and Section 8.4 gives illustrative sample cases. Section 8.5 provides our prototypical tool support. Section 8.6 presents the evaluation of the presented approach, and Section 8.7 discusses the threats to the validity of our research. Finally, we conclude in Section 8.8.

Table 8.1.: The Mathematical Notations Used in this Chapter

Notation	Description
n_{rout}	Number of routers
n_{serv}	Number of services
n_{scal}	Number of scaling replicas
n_{pro}	Number of processing rate improvements
IR	Number of incoming requests
r/s	Requests per second
R	Reliability Request loss model in r/s
R_{th}	Reliability threshold in r/s
$RGain$	Reliability gain in %
$R_{n_{rout}}$	Reliability prediction in r/s
R_c	Reliability of an evaluation case c
P	Performance model in ms
P_{th}	Performance threshold in ms
$PGain$	Performance gain in %
$P_{n_{rout}}$	Performance prediction in ms
P_c	Performance of an evaluation case c
T	Observed system time in s
CI	Crash interval in s
cf	Incoming call frequency in r/s
Com	Set of all components
$Rout$	Set of all routers
d_c	Average downtime of a component c in r/s
CP_c	Crash probability of a component c in %
BFR	Buffer fill rate in r/s
BFR_r	Buffer fill rate of a router r in r/s
$BFR(n_{scal}, n_{pro})$	Buffer fill rate for autoscaling in r/s
μ	Processing rate of a component in r/s
μ_r	Processing rate of a router r in r/s
λ	Arrival rate of a component in r/s
λ_r	Arrival rate of a router r in r/s
$\overline{RR}$	Average reconfiguration ratio
$\overline{C}$	Average reconfiguration costs in $cents/s$
C_{th}	Cost Threshold in $cents/s$
$C(n_{rout})$	Reconfiguration costs in $cents/s$
$C(n_{scal} = 1)$	Cost of scaling out in $cents/s$
$C(n_{pro} = 1)$	Cost of increasing the processing rate in $cents/s$
$C(n_{scal}, n_{pro})$	Cost of multidimensional autoscaling in $cents/s$
$Cases$	Set of experiment cases
n_c	length of $Cases$

8.2. Approach Overview

This section overviews our approach regarding the multifaceted reconfiguration of dynamic routing applications. Table 8.1 presents the mathematical notations used in this chapter.

Remember that a *router* is defined as an abstraction for any controller component that makes routing decisions, e.g., an API gateway [79], an enterprise service bus [26], or sidecars [49] (see Chapter 2). We model the system components, i.e., services and routers, as queuing stations[51] having two subcomponents, a buffer and a processor, as shown in Figure 8.1. Let λ be the arrival rate and μ the processing rate of a component based on the number of requests per second r/s. Incoming requests are buffered in a queue by a rate of λ and processed by a rate of μ.

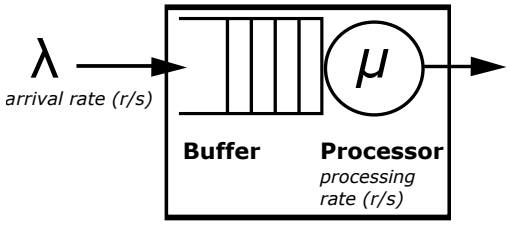


Figure 8.1.: Components as Queuing Stations

A component is in a *steady state* when its processing rate is greater than or equal to its arrival rate:

$$\mu \geq \lambda \tag{8.1}$$

In the steady state, a component is not overloaded and can process incoming requests without delay because of buffering. On the other hand, the *transient state* refers to when a component is overloaded because its processing rate is lower than the arrival rate of the requests:

$$\mu < \lambda \tag{8.2}$$

We study three interrelated scenarios for components:

- when components are idle and can be depleted.
- when components are active and steady.
- when components are overloaded.

The first scenario considers the infrastructure changes when a reconfiguration occurs. The second scenario studies a *per system* reconfiguration, which means we monitor the state of a system as a whole and reconfigure the components. The third scenario is a *per component* reconfiguration, i.e., our approach monitors and reconfigures each component separately.

8.3. Approach Details

This section presents the details of our approach regarding the investigating scenarios in this chapter.

8.3.1. Depletion of Idle Components

Some containerized system components process requests sporadically, so we deplete these containers when idle (see Chapter 7). However, the depleted containers can become active again and must be rescheduled. So we must consider the infrastructure changes, e.g., not overloading cloud nodes. We use an IaC component to automatically create and free cloud nodes to efficiently use resources and reduce costs. On the one hand, when depleting idle containers, efficiently rescheduling other active containers might free a node. On the other hand, when a depleted container receives a request and needs to be rescheduled, all nodes might be occupied. IaC mponent can create a node and schedule the container.

Assume the capacity of each node is known based on the number of scheduled containers. Algorithm 5 provides the steps to reconfigure the infrastructure. The IaC component calculates the total capacity of nodes. If the number of system components exceeds the total capacity, a new node is created, and containers are scheduled. Otherwise, IaC checks if the containers can be rescheduled efficiently on fewer nodes. Having defined the reconfiguration steps, we check if a container is idle and deplete it. When a request is received for a depleted container, ADR schedules it either on existing nodes or a new one.

Algorithm 5: Infrastructure Reconfiguration Algorithm (reconfigure)

```
Input: n_serv, n_rout
totalCapacity ← 0
foreach node : nodes do
    totalCapacity ← totalCapacity + capacity(node)
end
scheduleContainers()
if (n_serv + n_rout) > totalCapacity then
    createNode()
    scheduleContainers()
else
    foreach node : nodes do
        restCapacity ← totalCapacity − capacity(node)
        if (n_serv + n_rout) ≤ restCapacity then
            scheduleContainers()
            deleteNode(node)
        end
    end
end
```

8.3.2. Reconfiguration of Steady Components

When all components are active and steady according to Equation (8.1), we consider a system-wide Multi-Criteria Optimization (MCO) [4] optimizing reliability and performance trade-offs of the system.

Definitions

Remember that we defined the following elements in our reliability and performance models: n_{rout} and n_{serv} are the number of routers and services of an ADR application. CI is the crash interval, i.e., the interval during which we check for a crash of a component. Assuming the heartbeat pattern [46] or the health check API pattern [76] are used, CI is the time between two consecutive health checks. cf is the call frequency based on requests per second (r/s). Com is the set of components, i.e., routers and services. CP_c is the crash probability of each component, and d_c is the average downtime of a component c after it crashes.

Reliability Model

Let R be reliability. Based on Bernoulli processes [90], we model request loss during component crashes as explained in Chapter 3.

$$R = \frac{\lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} CP_c \cdot d_c}{T} \tag{8.3}$$

In this formula, request loss is defined as the number of client requests not processed due to a failure, such as a component crash. Equation (8.3) gives the request loss per second as a metric of reliability by calculating the expected value of the number of crashes. Having this information, we sum all the requests received by a system during the downtime of a component and divide them by the observed system time.

Performance Model

Let P be performance. We model the average processing time of requests per router as a performance metric. This metric is important as it allows us to study the quality of service factors, e.g., the efficiency of architecture configurations as elaborated in Chapter 4.

$$P = \frac{T}{n_{rout} \cdot cf \left(T - \lfloor \frac{T}{CI} \rfloor \cdot \sum_{c \in Com} CP_c \cdot d_c\right)} \tag{8.4}$$

We count the processed requests in this formula by subtracting the request loss from the total requests. We divide the observed time by the processed requests and the number of routers. Section 8.4.1 presents an illustrative sample case.

System-Wide MCO

We perform a multi-criteria optimization analysis to reconfigure an application by adjusting n_{rout}. We use the notations $R_{n_{rout}}$ and $P_{n_{rout}}$ to specify the reliability and performance predictions of an ADR configuration by their number of routers. Let R_{th} and P_{th} be the reliability and P_{th} performance thresholds. We define $C(n_{rout})$ as the reconfiguration costs for an architecture configuration and C_{th} as the cost threshold. We must ensure that we do not overload the routers. Let *Rout* be the set of routers of a system:

$$
\begin{aligned}
&Minimize \\
&\quad R_{n_{rout}} && (8.5)\\
&\quad P_{n_{rout}} && (8.6)\\
&Subject\ to \\
&\quad R_{n_{rout}} \leq R_{th} && (8.7)\\
&\quad P_{n_{rout}} \leq P_{th} && (8.8)\\
&\quad C(n_{rout}) \leq C_{th} && (8.9)\\
&\quad 1 \leq n_{rout} \leq n_{serv} && (8.10)\\
&\quad \mu_r \geq \lambda_r \quad \forall\ r \in Rout && (8.11)
\end{aligned}
$$

Algorithm 6: System-Wide Optimization Analysis (systemWideMCO)

Input: cf, n_{serv}, R_{th}, P_{th}, C_{th}

highBound $\leftarrow$ ($R = R_{th}$) and ($1 \leq$ highBound $\leq n_{serv}$)

lowBound $\leftarrow$ ($P = P_{th}$) and ($1 \leq$ lowBound $\leq$ highBound)

routersRange $\leftarrow$ {}

foreach *solution : [lowBound, highBound]* **do**

  solutionInRange $\leftarrow$ true

  if $C(n_{rout}) > C_{th}$ **then**

    solutionInRange $\leftarrow$ false

  end

  foreach *r : Rout* **do**

    if $\lambda_r > \mu_r$ **then**

      solutionInRange $\leftarrow$ false

  end

  if *solutionInRange* **then**

    routersRange $\leftarrow$ *routersRange* $\cup$ {*solution*}

  end

end

return routersRange

In the aforementioned system-wide MCO, we aim to minimize request loss and average processing time of requests per router without the prediction values violating R_{th} and P_{th}. Typically, there is no single answer to an MCO problem but a set of acceptable points in the solution space [4]. Algorithm 6 provides a simple solution to find a range of acceptable n_{rout}. The lower end of this range represents more centralized routing, so we find the lowest acceptable n_{rout} that does not violate the performance threshold. Conversely, the highest possible n_{rout} is bound by the reliability threshold. Having found the lower and upper values, we exclude the solutions that violate the cost threshold or result in overloading a router.

Preference Function

We must choose a final reconfiguration solution on the n_{rout} range returned from the above analysis. An architect assigns weights to reliability and performance, so a preference function can automatically choose a final solution. For example, when performance is highly important, the preference function selects a higher n_{rout} to choose more distributed routing. This reconfiguration processes requests in parallel, giving higher performance.

Reconfiguration Algorithm

Algorithm 7 presents our reconfiguration steps triggered, for instance, whenever reliability or performance metrics degrade. Time intervals, manual triggering, or changes in the incoming load can also trigger the algorithm if more appropriate than metrics degradation.

Algorithm 7: System-Wide Reconfiguration Steps (systemWideReconfig)

```
Input: R_th, P_th, C_th, performanceWeight
cf, n_serv ← consumeMonitoringData()
routersRange ← systemWideMCO(cf, n_serv, R_th, P_th, C_th)
reconfigSolution ← preferenceFunction(routersRange, performanceWeight)
reconfigure(n_serv, reconfigSolution)

function preferenceFunction(range, PW)
begin
    length ← max(range) - min(range) +1
    floor ← ⌊ PW * length ⌋
    if floor == max(range) then
        return max(range)
    else if floor == 0 then
        return min(range)
    else
        return floor + min(range) -1
    end
end
```

8.3.3. Autoscaling of Overloaded Components

When a system component is in a transient state following Equation (8.2), request processing is delayed because of buffering in an overloaded component. In this case, we use multidimensional autoscaling[1] to bring the transient component to a steady state. We consider two reconfiguration measures in a per-component MCO analysis, i.e., horizontal autoscaling (scaling out the component) and vertical autoscaling (adding resources to the component).

Buffer Fill Rate

Let BFR be the Buffer Fill Rate, defined as the difference between the arrival and processing rates.

$$BFR = \lambda - \mu \tag{8.12}$$

This metric is an indicator that a component is in a transient state. In this case, we reconfigure an overloaded component. Let n_{scal} be the number of scaling replicas, n_{pro} the number of processing rate improvements, $BFR(n_{scal}, n_{pro})$ the buffer fill rate predictions for multidimensional autoscaling, and IR the number of incoming requests for a component.

$$BFR(n_{scal}, n_{pro}) = \frac{IR \cdot cf}{n_{scal} + 1} - (\mu + n_{pro}) \tag{8.13}$$

This formula comes from the fact that scaling out an overloading component divides its arrival rate by the total number of replicas $(n_{scal} + 1)$. The BFR is also affected by the added processing rate $(\mu + n_{pro})$ as explained in Chapter 6.

Reconfiguration Cost

The cost of reconfiguration must be considered as well. Let $C(n_{scal}, n_{pro})$ be the cost of multidimensional autoscaling, $C(n_{scal} = 1)$ the cost of scaling out a component by one replica, and $C(n_{pro} = 1)$ the cost of increasing the processing rate of an overloading component by one r/s. The reconfiguration cost depends on the n_{scal} and n_{pro} improvements.

$$C(n_{scal}, n_{pro}) = n_{scal} \cdot C(n_{scal} = 1) + n_{pro} \cdot C(n_{pro} = 1) \tag{8.14}$$

Section 8.4.2 presents a parameterization and a sample case.

Reconfiguration Algorithm

Algorithm 8 presents the reconfiguration steps to autoscale a transient component.

[1] https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling

Algorithm 8: Multifaceted Reconfiguration Algorithm for an Overloading Component

Input: R_{th}, P_{th}, C_{th}, performanceWeight

$cf, n_{serv}, IR, \mu \leftarrow$ **consumeMonitoringData()**

paretoFront $\leftarrow$ **perComponentMCO**(cf, IR, μ, C_{th})

reconfigSolution $\leftarrow$ **preferenceFunction**(paretoFront)

reconfigure(n_{serv}, reconfigSolution)

systemWideReconfig$(R_{th}, P_{th}, C_{th},$ performanceWeight$)$

function preferenceFunction(paretoFront)
begin
  $C \leftarrow C_{th}$
  reconfigSolution $\leftarrow (0, 0)$
  foreach *solution : paretoFront* **do**
    $C(n_{scal}, n_{pro}) \leftarrow n_{scal} \cdot C(n_{scal} = 1) + n_{pro} \cdot C(n_{pro} = 1)$
    if $C(n_{scal}, n_{pro}) \leq C$ **then**
      $C \leftarrow C(n_{scal}, n_{pro})$
      reconfigSolution $\leftarrow$ solution
    end
  end
  return reconfigSolution
end

Per-Component MCO

We adjust the buffer fill rate of an overloading component to bring it to a steady state. This reconfiguration is based on a second multi-criteria optimization analysis performed for each component separately. We aim to minimize BFR but with a minimum reconfiguration cost given by the following formulae:

$$\begin{aligned} &Minimize \\ &\quad BFR(n_{scal}, n_{pro}) && (8.15)\\ &\quad C(n_{scal}, n_{pro}) && (8.16)\\ &Subject\ to \\ &\quad \lambda \leq \mu && (8.17)\\ &\quad C(n_{scal}, n_{pro}) \leq C_{th} && (8.18) \end{aligned}$$

Remember that there is typically no single answer to an MCO problem but a set of acceptable points called the Pareto front [4]. Using a preference function, we choose a final solution that brings the component to a steady state according to Equation (8.1) with a minimum cost. Having done this analysis for overloaded components, all system components are in a steady state. We must perform a system-wide MCO as described in Section 8.3.2.

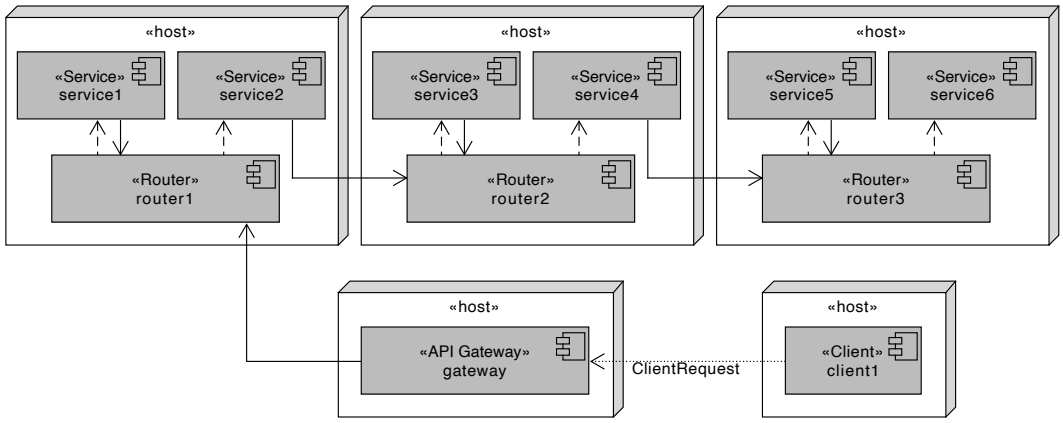


Figure 8.2.: Example Configuration of Dynamic Routing Applications (solid arrows show the incoming requests of routers.)

8.4. Illustrative Sample Cases

We provide a parameterization of our models alongside illustrative sample cases of steady and transient components.

8.4.1. Reconfiguration of Steady Components

We study an example from the data set of our experiment[6] to parameterize our models and give sample cases (see Section 2.3). An example configuration, where clients send requests to an API gateway that forwards them to the services, is shown in Figure 8.2. We observed the system for $T = 600\,s$, had a crash interval of $CI = 15\,s$ and studied uniform crash probabilities and downtimes for all components as $CP_c = 0.5\%$ and $d_c = 3\,s$. We can parameterize our reliability model (r/s) and performance model (ms) in Equations (8.3) and (8.4) as the following:

$$R = cf \cdot 0.001(n_{serv} + n_{rout}) \tag{8.19}$$

$$P = \frac{1000}{n_{rout} \cdot cf(1 - 0.001(n_{serv} + n_{rout}))} \tag{8.20}$$

In the example configuration, we have $n_{rout} = 3$ routers and $n_{serv} = 6$ services. Let us consider that this sample case has an expected call frequency of $cf = 25\,r/s$, and all routers have a processing rate of $\mu = 64\,r/s$. We parameterize the arrival rates and the number of incoming requests of routers (solid arrows in Figure 8.2) to check if they are overloaded. In our experiment, we allocated services equally to routers:

$$IR = \frac{n_{serv}}{n_{rout}} \tag{8.21}$$

$$\lambda_r = cf \cdot IR = \frac{cf \cdot n_{serv}}{n_{rout}} \quad \forall \quad r \in Rout \tag{8.22}$$

In our sample case, $IR = 2$ and $\lambda_r = 50\, r/s$. Therefore, all routers are steady according to Equation (8.1). To parameterize the cost functions, we use the Google Autopilot pricing[2]. Autopilot allows increments of 0.25 vCPUs per container (same is offered by Amazon Fargate[3]) that corresponds to 8 r/s in our experiment:

$$C(n_{pro} = 8) = 5 \cdot 10^{-4}\ cents\,/s \tag{8.23}$$

The scaling cost of our routers with $\mu = 64\, r/s$ accounts to:

$$C(n_{scal} = 1) = 4 \cdot 10^{-3}\ cents\,/s \tag{8.24}$$

Regarding thresholds, we consider a reliability threshold of 1.2 r/s, a performance threshold of 35 ms, and a cost threshold of 1 $cent/s$. We study a case where an architect assigns a weight of 1.0 to performance and 0.0 to reliability. We perform the system-wide MCO analysis in Section 8.3.2 by rewriting Equations (8.19) and (8.20) for these values:

Minimize

$$R_{n_{rout}} = 0.075 + 0.025 \cdot n_{rout} \tag{8.25}$$

$$P_{n_{rout}} = \frac{1000}{n_{rout} \cdot (24.925 - 0.025 \cdot n_{rout})} \tag{8.26}$$

Subject to

$$R_{n_{rout}} \leq 1.2 \tag{8.27}$$

$$P_{n_{rout}} \leq 35\ ms \tag{8.28}$$

$$C(n_{rout}) \leq 1\ cent/s \tag{8.29}$$

$$1 \leq n_{rout} \leq 6 \tag{8.30}$$

$$\mu_r \geq \lambda_r \quad \forall\ r \in Rout \tag{8.31}$$

Equation (8.25) informs that the reliability predictions in the $1 \leq n_{rout} \leq 6$ always satisfy the reliability threshold. In Equation (8.26), the constraint on the performance threshold of $P_{n_{rout}} \leq 35\ ms$ gives the lowest value for the number of routers as $n_{rout} = 2$. Therefore, the range for n_{rout} is:

$$2 \leq n_{rout} \leq 6 \tag{8.32}$$

Following Algorithm 6, we see that the cost threshold of 1 $cent/s$ is always satisfied in this range. We check if any solution results in overloading the routers in this range. On the lowest bound, i.e., $n_{rout} = 2$, we have the following according to Equations (8.21) and (8.22):

$$IR = 3 \tag{8.33}$$

$$\lambda_r = 75\ r/s \tag{8.34}$$

[2] https://cloud.google.com/kubernetes-engine/pricing
[3] https://aws.amazon.com/fargate/pricing/

Since $\mu_r = 64\,r/s$, this overloads the routers according to Equation (8.1). So we exclude this solution, and all the other points on the range are acceptable. Therefore, the acceptable n_{rout} range is the following:

$$3 \leq n_{rout} \leq 6 \tag{8.35}$$

The performance weight is 1.0, so the preference function chooses the highest possible value for n_{rout} according to Algorithm 7. Therefore, the final solution is a configuration with six routers, i.e., $n_{rout} = 6$. We use this analysis also when illustrating our other scenario, i.e., the multidimensional autoscaling of transient components to prevent system overload.

8.4.2. Autoscaling of Overloaded Components

Let us consider the studied example in Figure 8.2. Assume this application is stressed with a call frequency of $cf = 100\,r/s$. According to Equations (8.13), (8.21) and (8.22), we have the following:

$$IR = 2 \tag{8.36}$$
$$\lambda_r = 200\ r/s \quad \forall \quad r \in Rout \tag{8.37}$$
$$\mu_r = 64\ \ r/s \quad \forall \quad r \in Rout \tag{8.38}$$
$$BFR_r = 136\ r/s \quad \forall \quad r \in Rout \tag{8.39}$$

Having the same cost threshold of $C_{th} = 1\ cents/s$, we can rewrite the per-component MCO analysis in Section 8.3.3:

$$Minimize$$
$$\frac{200}{n_{scal}+1} - (n_{pro} + 80) \tag{8.40}$$
$$8 \cdot n_{pro} \cdot 5 \cdot 10^{-4} + n_{scal} \cdot 4 \cdot 10^{-3} \tag{8.41}$$
$$Subject\ to$$
$$\lambda \leq \mu \tag{8.42}$$
$$C(n_{scal}, n_{pro}) \leq 1\ cents/s \tag{8.43}$$

We choose a final solution that brings the component to a steady state with a minimum cost. Following Algorithm 8, this reconfiguration solution is:

$$(n_{scal}, n_{pro}) = (1, 40) \tag{8.44}$$

which gives the buffer fill rate of $BFR(1, 40) = -4$. This solution results in scaling out each router and increasing n_{rout} from three to six routers. Therefore, we must check that the system-wide MCO does not violate the thresholds. As we calculated before in Equation (8.35), the acceptable range of routers is $3 \leq n_{rout} \leq 6$. So the solution is acceptable.

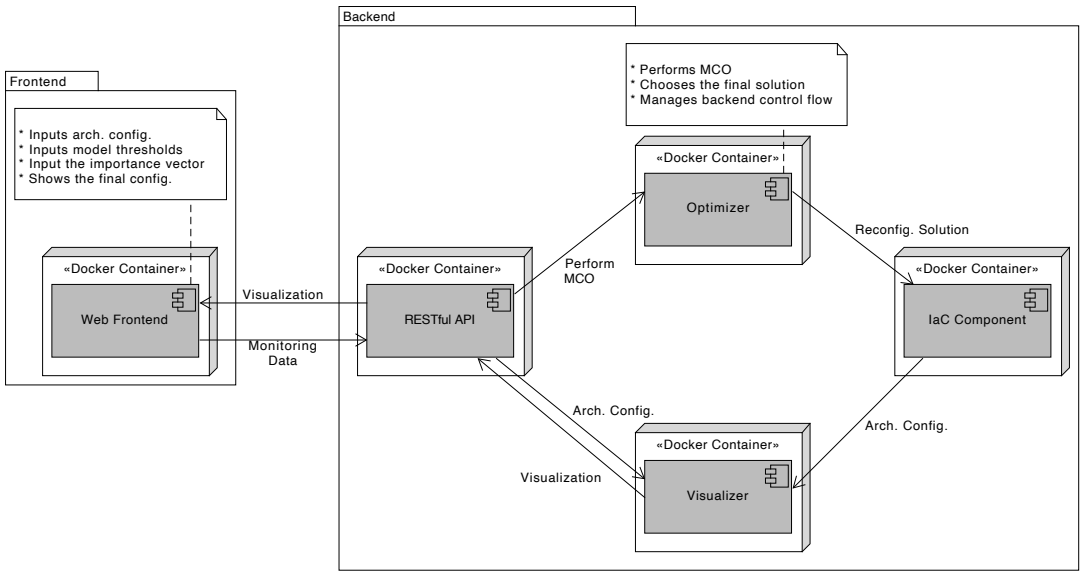


Figure 8.3.: Tool Architecture Diagram

8.5. Tool Support

We provide a prototypical tool to demonstrate our approach in the online artifact of this thesis[6].

8.5.1. Tool Architecture

Figure 8.3 shows the high-level tool architecture. We have a slightly different architecture than the one presented in Figure 5.1. Because we study multifaceted optimization of dynamic routing in this chapter, we decided to define an *Optimizer* container. The *Web Frontend* of our application provides functionalities to specify architecture configurations and model elements, such as thresholds. This information is sent to the *RESTful API* in the backend that invokes the *Optimizer* to perform MCO analyses and find the final reconfiguration solution. The *IaC Component* generates artifacts in the form of Bash[4] scripts and configuration files, e.g., infrastructure configuration data to be used by an IaC tool. These scripts can be used to schedule containers using the Docker technology[5]. The *Visualizer* creates diagrams of the configurations using PlantUML[6] that are shown in the *Web Frontend*. The frontend is implemented in React[7] and the backend in Node.js[8].

[4] https://www.gnu.org/software/bash/
[5] https://www.docker.com/
[6] https://plantuml.com/
[7] https://reactjs.org/
[8] https://nodejs.org/

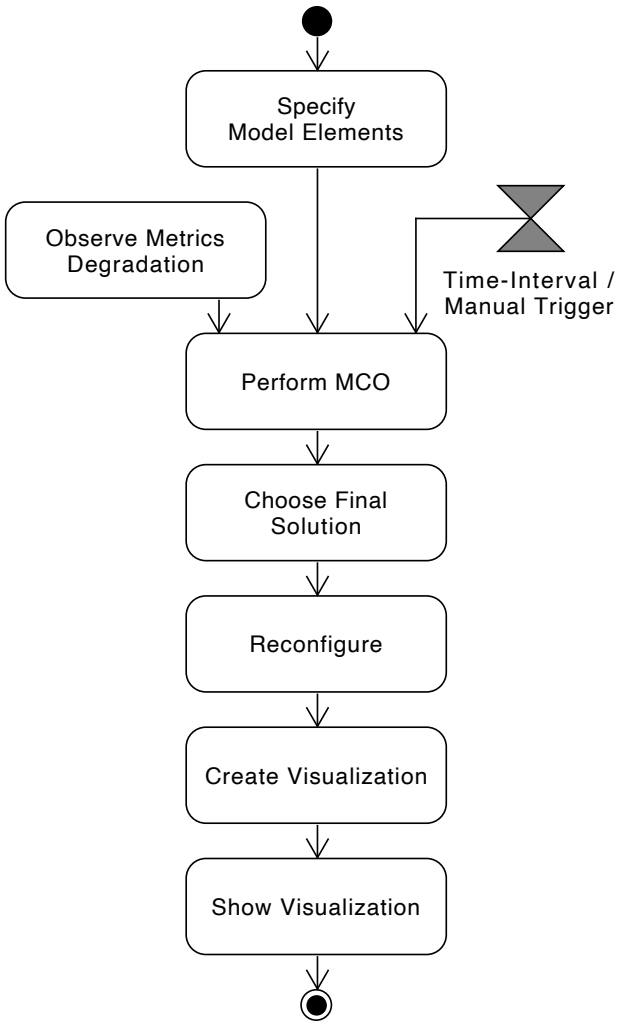


Figure 8.4.: Model Reconfiguration Toolflow

8.5.2. Toolflow

Figure 8.4 shows the flow regarding the model reconfiguration. An architect specifies various model elements, i.e., the number of routers and services, thresholds, incoming call frequencies, performance weight, processing rates of components, and cost functions. A reconfiguration is triggered when metrics degradation is observed, according to timers or manually. When reconfiguration is triggered, the backend performs an MCO analysis, chooses a final reconfiguration solution, and generates IaC artifacts. The reconfiguration visualization is then created using PlantUML and shown in the frontend.

8.6. Evaluation

This section evaluates our approach in both scenarios illustrated in Section 8.4 systematically. We compare the model values to our empirical data set reported in the online artifact of this dissertation[6]. Note that our study is neither specific to our experiment infrastructure nor our cases. We use our empirical data set to evaluate our approach using measured data from an extensive experiment (see Section 2.3.2 for details).

8.6.1. Reconfiguration of Steady Components

This section presents our evaluation of when components are steady.

Evaluation Cases

We systematically evaluate our method through various thresholds and importance weights for reliability and performance. We compare our model predictions with 9 experiment cases, i.e., three levels of routers and three levels of services, each operational for four levels of cf.

$$n_{serv} \in \{\ 3, 5, 10\ \} \tag{8.45}$$
$$n_{rout} \in \{\ 1, 3, n_{serv}\ \} \tag{8.46}$$
$$cf \in \{\ 10, 25, 50, 100\ \}\ r/s \tag{8.47}$$

Regarding the cost thresholds, we take $C_{th} = 1\,cent/s$ as in our illustrative sample cases. For the processing rates, we investigate 9 levels as follows. In Section 8.4.1, we mentioned that a component with one vCPU has a processing rate of roughly $32\,r/s$ in our experiment. We start with components having two vCPUs up to six in increments of 0.5 vCPUs.

$$64 \leq \mu \leq 192\ r/s \tag{8.48}$$

Regarding reliability and performance thresholds, we start with tight reliability and loose performance thresholds so that more centralized routing is acceptable (lower value of n_{rout}). We increase the reliability and decrease the performance thresholds by 10% in each step so that distributed routing becomes applicable. To find the starting points, we consider the worst-case scenario of our empirical data. Equation (8.3) informs that a higher n_{serv} results in a higher expected request loss as the number of components increases. In our experiment, the highest number of services is ten. With $n_{serv} = 10$, the worst-case reliability for centralized routing and fully distributed routing ($n_{rout} = 10$) is 1.1 and 2.0 r/s, respectively.

Regarding performance, for the case of $n_{serv} = 10$, we investigate our predictions to find a range where a reconfiguration is possible. The lowest possible performance prediction is 33.7 ms, and the highest is 101.1 ms. We adjust these values slightly and take our

boundary thresholds as follows. We analyze step-by-step by increasing the reliability threshold and decreasing the performance threshold by 10% as before.

$$1.1 \leq R_{th} \leq 2.0 \ r/s \tag{8.49}$$

$$35 \leq P_{th} \leq 100 \ ms \tag{8.50}$$

Results Analysis

We evaluate 9801 systematic evaluation cases: 9 experiment cases, 9 processing rate levels, 11 importance weights, and 11 thresholds. We define reliability gain, i.e., $RGain$, and performance gain, i.e., $PGain$, as the average percentage differences of our predictions compared to those of fixed architectures. Let R_c and P_c be the reliability and performance of an evaluation case c, $Cases$ the set of experiment cases, and n_c the length of $Cases$. Note that the following formulae are based on the Mean Absolute Percentage Error (MAPE) widely used in the cloud quality-of-service research [90].

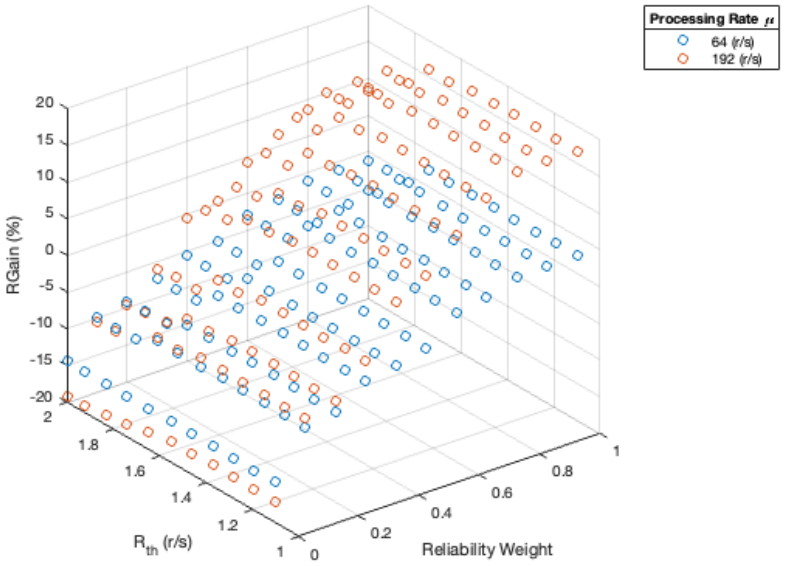
$$RGain = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{R_c - R_{n_{rout}}}{R_{n_{rout}}} \tag{8.51}$$

$$PGain = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{P_c - P_{n_{rout}}}{P_{n_{rout}}} \tag{8.52}$$

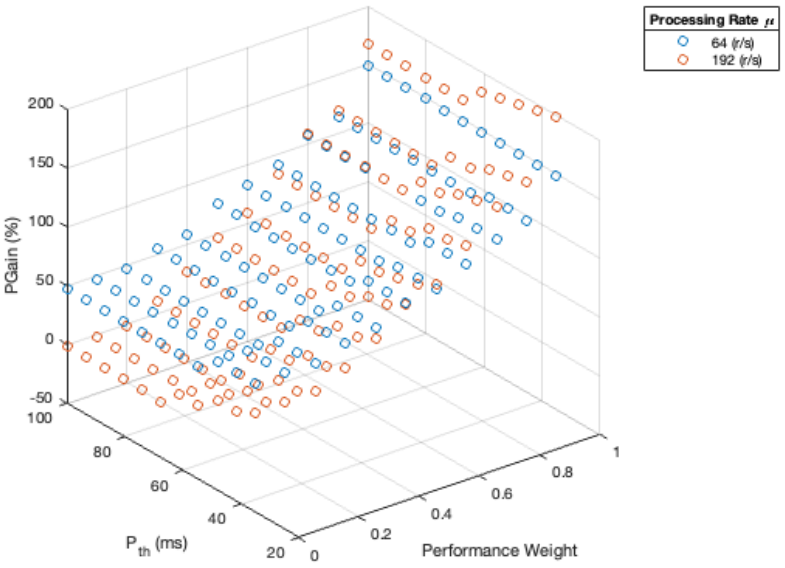
Remember $R_{n_{rout}}$ and $P_{n_{rout}}$ are reliability and performance predictions. The gains are averaged over $n_c = 9$ experiment cases.

Figure 8.5 shows the reliability and performance gains. Moreover, each figure shows the plots for our lowest studied processing rate of $\mu = 64\,r/s$ and the highest bound in our research, i.e., $\mu = 192\,r/s$. Regarding reliability, we can see in Figure 8.5a that with a higher reliability weight, we have an increase in reliability gain with $\mu = 192\,r/s$. Remember in Algorithm 6, we check that the components are not overloaded when choosing a more centralized routing to increase reliability. Having a higher processing rate results in a component processing higher call frequencies without being overloaded. However, as a result of choosing a less centralized routing, the gain in reliability is at most 16.60%.

Our approach provides significant improvements in performance gains. As more importance is given to the performance of a system, i.e., performance weight increases, our approach reconfigures an application by choosing more distributed routing. This reconfiguration results in a rise of a performance gain as shown by Figure 8.5b. On average, when cases with correct and incorrect architecture choices are analyzed together, our adaptive method provides 74.22% performance gain. A higher gain for performance compared to reliability is expected. To clarify, Equations (8.19) and (8.20) inform that changing the number of routers has a higher effect on the performance than a system's reliability. We define performance as the average processing time of requests per router. Having a higher number of routers to process the requests in parallel results in dividing the average processing time by more routers.

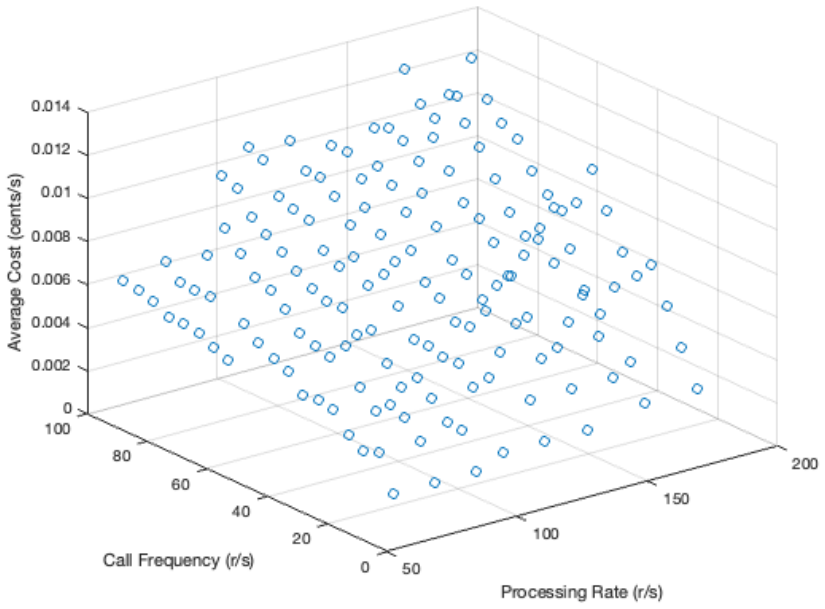


(a) Reliability Gain

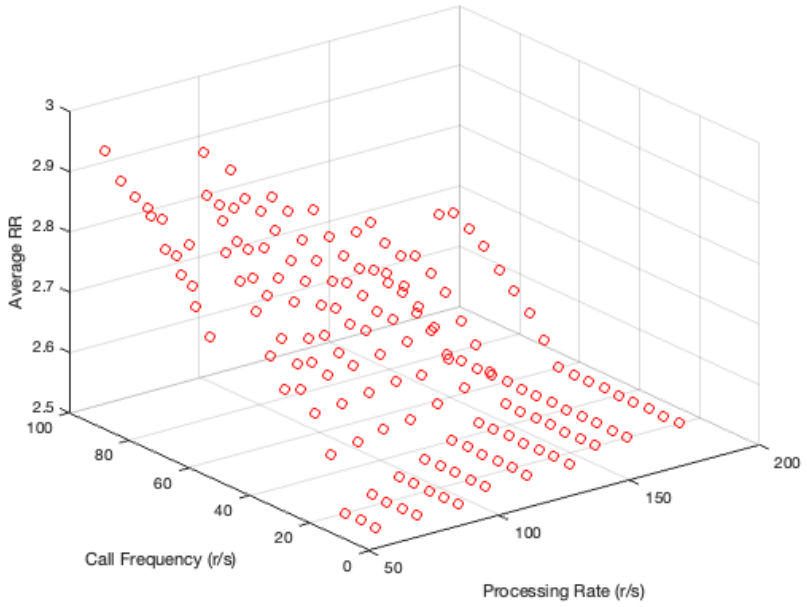


(b) Performance Gain

Figure 8.5.: Reliability and Performance Gains with Processing Rates of $\mu = 64$ and $192\ r/s$



(a) Average Cost



(b) Average Reconfiguration Ratio

Figure 8.6.: Plots of Evaluation Data for the Autoscaling of Transient Components

8.6.2. Autoscaling of Overloaded Components

This section evaluates our approach systematically regarding component overload prevention.

Evaluation Cases

We go through the same range of values as in the previous scenario:

$$n_{serv} \in \{\ 3, 5, 10\ \} \tag{8.53}$$
$$n_{rout} \in \{\ 1, 3, n_{serv}\ \} \tag{8.54}$$
$$10 \leq cf \leq 100\ \ r/s \tag{8.55}$$
$$64 \leq \mu \leq 196\ \ r/s \tag{8.56}$$
$$C_{th} = 1\ cent/s \tag{8.57}$$

As before, we study increments of 0.5 vCPUs resulting in 9 μ levels. We consider the same range of call frequencies. However, since we are studying component overloads, we evaluate increments of 5 r/s, resulting in 19 cf levels.

Results Analysis

We evaluated 1539 cases for this scenario, i.e., three levels of n_{serv}, three levels of n_{rout}, 9 levels of μ, and 19 levels of cf. We define the average cost $\overline{C}$ and the average reconfiguration ratio $\overline{RR}$, that is, the amount of BFR improvements per cost spent as:

$$\overline{C} = \frac{1}{n_c} \cdot \sum_{c \in Cases} C(n_{scal}, n_{pro}) \tag{8.58}$$

$$\overline{RR} = \frac{1}{n_c} \cdot \sum_{c \in Cases} \frac{BFR(0,0) - BFR(n_{scal}, n_{pro})}{C(n_{scal}, n_{pro})} \tag{8.59}$$

$BFR(0,0)$ is the buffer fill rate without reconfiguration. We average over three levels of services and three levels of routers, i.e., $n_c = 9$.

In Figure 8.6a, we can see that the reconfiguration costs increase as the processing rate and the call frequency goes higher. This is expected as reconfiguring a component with a higher processing rate is more expensive, especially when scaling out an overloaded component. Moreover, a higher incoming call frequency results in more overloaded components and, consequently, higher reconfiguration costs. However, as seen in Figure 8.6b, a higher cf results in a higher reconfiguration ratio. Our approach balances the costs with a bigger buffer fill rate improvement converging $\overline{RR}$ to an average of 2.62. The average reconfiguration cost over all cases is 0.0065 $cents/s$, bringing all overloading system components to a steady state.

8.7. Threats to Validity

There are several threats to the validity and limitations of our study that we discuss in this section based on the four threat types by Wohlin et al. [96].

8.7.1. Construct Validity

We used request loss and the average processing time of requests per router as reliability and performance metrics, respectively. The threat remains that other metrics might model these quality attributes better, e.g., a cascade of calls beyond a single call sequence for reliability [67], or data transfer rates of messages which are m byte-long for performance [53]. Moreover, we studied reconfiguration measures of increasing the processing rate and scaling out a component to prevent system overload. While this is a common approach in service- and cloud-based research (see state of the art in Section 1.3), other measures might work better in terms of system overload prevention, for instance, changing the routing technology or using a circuit breaker [78]. More research with real-world systems is required to mitigate this threat.

8.7.2. Internal Validity

Internal validity concerns factors that affect the independent variables concerning causality. We considered a simple reconfiguration strategy to start the new setup in parallel with the running configuration to avoid impacts on reliability (e.g., request loss due to reconfiguration) and performance (e.g., increased processing time while reconfiguring). In a real-world system, this solution is cost-ineffective that introduces additional resource demands. The architects must specify a reconfiguration strategy based on their application needs to exclude this threat. Moreover, we only considered constant load when modeling the stress of components using queuing theory. In reality, cloud-based systems are met with different load profiles, e.g., sudden load spikes.

8.7.3. External Validity

External validity concerns threats that limit the ability to generalize the results beyond the experiment. We designed our novel architecture with generality in mind. However, the threat remains that evaluating our approach based on another infrastructure may lead to different results. To mitigate this thread, we evaluated our proposed approach with an extensive systematic evaluation using the data of our experiment of 1200 hours (see Section 8.6). Moreover, the results might not be generalizable beyond the given experiment cases of 10-100 requests per second and call sequences of length 3-10. As this covers a wide variety of loads and call sequences in cloud-based applications, the impact of this threat should be limited.

8.7.4. Conclusion Validity

Conclusion validity concerns factors that affect the ability to conclude the relations between treatments and study outcomes. As the statistical method to evaluate the accuracy of our prediction models, we used the Mean Absolute Percentage Error (MAPE) metric [90]. We defined reliability and performance gains, as well as the average percentage difference of buffer fill rate based on MAPE, as it is widely used and offers good interpretability in our research context.

8.8. Conclusions

In this chapter, we proposed a multifaceted reconfiguration approach that self-adapts between different routing patterns considering the component overloads and idleness. Moreover, we provided a prototypical tool that provides visualizations to study different architecture configurations. We systematically evaluated our approach based on our empirical data (see Section 8.2 for details). Our extensive systematic evaluation shows significant improvements in quality trade-off adaptations and system overload prevention. In our experiment cases, our approach can yield up to 16.60% reliability gain. On average, where cases with the right and the wrong architecture choices are analyzed together, our approach offers a 74.22% performance gain.

9. Conclusions and Future Work

In this dissertation, we investigated the modeling and multifaceted reconfiguration of cloud-based dynamic routing. Each chapter includes a conclusions section; hence we provide an overall thesis summary in this chapter. Moreover, we plan our future studies that use machine-learning techniques such as an Artificial Neural Network (ANN) [64] to learn the studied reconfiguration concepts. In this doctoral thesis, we investigated the following research questions:

- ***RQ*$_1$** *What elements are required to automatically assess the different quality-of-service requirements in cloud resource management of service- and cloud-based dynamic routing applications?*
- ***RQ*$_2$** *How to choose the final reconfiguration option as part of a feedback loop to manage cloud resources efficiently, and how well does this reconfiguration solution perform compared to the case when one architecture configuration runs statically?*
- ***RQ*$_3$** *What is the architecture of a supporting tool that analyses the system quality-of-service requirements and facilitates the reconfiguration of a dynamic routing application using the optimal configuration solution?*

For ***RQ*$_1$**, we provided reliability and performance models in Chapters 3 and 4. For ***RQ*$_2$**, we proposed the Adaptive Dynamic Routers (ADR) architecture and presented our approach details in Chapters 5 to 7. For ***RQ*$_3$**, we provided prototypical tool support for the multifaceted reconfiguration of dynamic routing applications in Chapter 8 (Chapter 5 also provides tool support).

Our study concludes that more decentralized routing results in losing more requests, i.e., lower reliability, compared to more centralized approaches. However, our results show that distributed settings indicate better performance, especially under high load, because of using more routers. The major impact of our work is on architectural design decisions for dynamic routing. Our proposed analytical models represent service- and cloud-based systems. They can be used in other environments and applications to give insight to architects when making architectural design decisions regarding dynamic routing. To the best of our knowledge, this work is the first to provide empirical evidence on the trade-off analysis between reliability and performance in cloud-based dynamic routing systems. The primary achievements of our research comprise models and an empirical examination of prevalent dynamic routing architectures. Such in-depth empirical studies establish a firm basis for comprehending the current state of the art and its constraints, defining fundamental facts, and delivering data sets for subsequent research, all of which are essential for the development of novel algorithms and architectures.

For this purpose, we provide an extensive data set of 36336300 points in our online artifact[6] based on the concepts introduced in this dissertation. The input of our data set is all combinations of the following six monitoring data. The output is the result of the multifaceted reconfiguration of dynamic routing as presented in Chapter 8. Remember that we studied the following model elements: n_{rout} and n_{serv} are the number of routers and services of an ADR application. cf is the call frequency based on requests per second (r/s). R_{th} is the reliability threshold based on the number of request losses per second. P_{th} is the performance threshold as the average processing time of requests per router. PW is the performance weight, i.e., a number between 0.0 and 1.0, giving the importance of performance compared to reliability. We have the following ranges in our data set, and the rationale behind choosing them is given below:

$$3 \leq n_{serv} \leq 10 \tag{9.1}$$
$$1 \leq n_{rout} \leq n_{serv} \tag{9.2}$$
$$10 \leq cf \leq 100\, r/s \tag{9.3}$$
$$1.1 \leq R_{th} \leq 2\, r/s \tag{9.4}$$
$$35 \leq P_{th} \leq 100\, ms \tag{9.5}$$
$$0.0 \leq PW \leq 1.0 \tag{9.6}$$

These values are based on an extensive experiment of 1200 hours, in which we measured the quality-of-service metrics of dynamic routing applications (see Section 2.3.2). The call frequency of $cf = 100\, r/s$, or even lower numbers, is chosen in many studies (see, e.g., [33, 87]). Therefore, we chose different portions between 10 to 100 r/s. As for the number of services n_{serv}, based on our experience and a survey on existing cloud applications in the literature and industry (see, e.g., [5, 11, 12]), the number of cloud services that are directly dependent on each other in a call sequence is usually rather low. As a result, we study 3 to 10 services in a call sequence. For the reliability and performance thresholds, we studied our empirical data and chose the worst-case scenarios for centralized and distributed routing (see Chapter 8). The importance weights are required to choose a final reconfiguration solution based on the need of different applications. For example, giving a higher weight to performance opts for more distributed routing because of the parallel processing of requests. A performance weight of 1.0 emphasizes performance, and 0.0 gives weight to reliability when choosing the final solution.

This data set can be used to train an ANN for the multifaceted reconfiguration of the dynamic-routing architectures. For our future work, we plan to investigate different ANN configurations in a series of studies. To this end, we have already published one research paper [8] analyzing a deep neural network [64] that has learned our data set. In this study, we used the TensorFlow[1] and the Keras sequential model[2]. A sequential model has exactly one input and one output. The introduced ANN has 5 densely-connected layers, each with 40 neurons. We used the standard rectifier-linear-unit activation function, the mean-squared-error loss function, and the Adam optimizer [23] with a learning rate of

[1] https://www.tensorflow.org/
[2] https://www.tensorflow.org/guide/keras/sequential_model

0.001. We trained the deep neural network with 100 epochs. The ANN and the paper are available in the online appendix of this dissertation. Moreover, we plan to combine reinforcement learning [63] with neural networks to predict architecture configurations, load profiles, infrastructure changes, etc.

A. Initial Performance Experiment

This appendix relates to C_2: *Performance models and trade-offs analysis*. In the research paper [5], we performed an initial experiment and studied statistical regression analyses of the empirical data. This paper was the foundation for our performance study [12], where we investigated a larger-scale scientific experiment presented in Chapter 4 of the dissertation.

A.1. Introduction

In many service-based applications, decisions about data routing need to be made at runtime, for instance, to ensure compliant data handling. In the context of software systems, compliant data handling is concerned with ensuring that the system handles data in accordance with established laws, regulations, and business policies. Different service- and cloud-based architectures to make dynamic data routing decisions exist, including central entities, multiple dedicated decision services, or using a sidecar for each involved service. These architectures differ in various quality attributes, including complexity, understandability, and changeability of the decision logic. A crucial concern in cloud-based settings with a high load of incoming requests is the performance of the chosen architecture, which is often hard to predict in early architecture designs. Choosing the wrong architecture for decision-making at runtime may severely impact the performance of the software system. In this appendix, we evaluate the performance of three representative approaches for processing compliance rules concerned with data routing in service- and cloud-based architectures (see Section 2.1). The results show that distributed approaches for dynamic data routing have a better performance compared to centralized solutions. On the other hand, centralized solutions are easier to understand and change, but this depends on the domain problem strongly. Our study provides prediction models to estimate the performance trade-offs an architect has to accept for improvement in other relevant qualities, such as the number of required cloud resources.

The structure of this appendix is as follows: Section A.2 explains our experiment planning. Section A.3 presents the analysis of the experimental results, and Section A.4 considers threats to validity. Finally, Section A.5 summarizes our findings and concludes this appendix.

A.2. Experimental Planning

In this section, we present the details of our initial experiment. Table A.1 presents the mathematical notations used in this appendix.

Table A.1.: The Mathematical Notations Used in this Appendix

Notation	Description
RTT	Round-trip time
RTT_n	Median round-trip time of a number of services
Q_1	First quartile of the recorded RTT values
Q_3	Third quartile of the recorded RTT values
σ	Standard deviation of the recorded RTT values
$WAvg$	Weighted Average
r/s	Requests per second
cf	Incoming call frequency
n_{serv}	Number of services
SC	Service coefficient
FC	Frequency coefficient
IC	Interaction coefficient
Int	Intercept

A.2.1. Goals

The experiment aims to measure the performance of the three approaches for processing compliance rules concerned with data routing in service- and cloud-based architectures, i.e., the Central Entity (CE), Dynamic Routers (DR), and the Sidecar-based Architecture (SA) as presented in Section 2.1.

A.2.2. Experiment Design

The experiment design presented in this study is slightly different from that of our main experiment (see Section 2.3.2).

Technical Details

We used a private cloud with four nodes, each having two identical CPUs. Two cloud nodes host Intel®Xeon®E5-2680 v4 @ 2.40GHz[1] and the other two host the same processor family but version v3 @ 2.50GHz. The v4 and v3 versions have 14 and 12 cores respectively and two physical threads per core (56 and 48 threads in total). All cloud nodes have 256 GB of system memory and run Ubuntu Server 18.04.01 LTS[2]. On top of the operating system, Docker[3] containerization is used to run the cloud services implemented using Node.js[4]. We utilized five desktop computers to simulate load generation, each hosting an Intel®Core™i3-2120T CPU @ 2.60GHz with two cores and two physical threads per core (four threads in total). All desktop computers have 8 GB of system memory and run

[1] https://www.intel.com/content/www/us/en/homepage.html
[2] https://www.ubuntu.com
[3] https://www.docker.com
[4] https://nodejs.org/en/

Ubuntu 18.10. They generate load using Apache JMeter[5] that sends Hypertext Transfer Protocol (HTTP) version 1.1[6] requests to cloud nodes.

Architecture Configurations

We used one cloud node with 56 threads to run the API gateway and distributed cloud services among the remaining three nodes. The services are distributed so that all nodes have the same number of cloud services (with a maximum difference of one service). In the case of CE, the central entity service is also placed on the node where the API gateway is scheduled to minimize network communication. For DR, we placed a router on each of the three nodes that host cloud services. Each bucket controls data communication regarding services on their corresponding node. We call this configuration *Dynamic Routers With Three Routers (DR_3)*. We added another configuration for DR in the sense that we put two routers on each cloud node and let each router control data flow for half of the cloud services on the corresponding node. We call this configuration *Dynamic Routers With Six Routers (DR_6)*. SA places one sidecar per each cloud service on the corresponding node. Additionally, we implemented the aforementioned configurations on a single-node environment to be able to compare the performance of architectures when deployed on a distributed or a local setup.

We chose to implement all three architecture options from scratch in Node.js and did not use existing implementations of these options, such as Envoy[7] for sidecar architectures. The reason is that we wanted comparable implementations to avoid measuring the impact of a particular technology implementation rather than the impact of the canonical architecture.

Round-Trip Time Calculation

Let RTT be the round-trip time. To measure the performance of the different prototypical architectures, we calculated the RTT of requests, which is defined as the difference in time from the moment a request enters the application through the API gateway until it is routed through all cloud services involved in the processing of the request. JMeter generates an identification (ID) number for each HTTP request. Whenever the gateway receives a request, it starts a timer with an attached ID. The request is routed through cloud services and returns to the gateway when processing is finished, i.e., either the request reaches its destination or the controlling logic cuts the data flow short before the request is processed by its final cloud service. Next, the gateway reads the request ID and stops the corresponding timer. The RTT is the time calculated by the timer.

[5] https://jmeter.apache.org
[6] https://tools.ietf.org/html/rfc7230
[7] https://www.envoyproxy.io/

Experiment Cases

Many factors can influence RTT, out of which we chose two to study their effects: The call frequency and the number of cloud services. Let cf be the incoming call frequency, which is defined as the number of requests per second coming from service clients. cf affects RTT since a higher frequency of calls requires either more processing power or buffering. Let n_{serv} be the number of services of a dynamic routing application. A higher n_{serv} increases RTT because there are more rules to be checked by controlling services.

In this experiment, we chose call frequencies of 100, 500, and 1000 requests per second (r/s). We selected these numbers based on a study of related works. In many related studies, 100 requests per second (or even lower numbers) are chosen (see, e.g. [33, 87]). Focusing on higher loads, we chose 100 r/s as our lowest studied call frequency. A recent benchmark for self-adaptive Infrastructure as a service (IaaS) cloud environment [43] uses 339 requests per second as its upper limit. We thus chose 500 r/s as a close but slightly higher number to focus on high-load scenarios. Finally, to study even higher load conditions, we took 1000 r/s into consideration. In the case of 100 r/s, one desktop computer generates the load. For the call frequencies of 500 and 1000 r/s, we used two and five computers, respectively.

We chose the experimental cases of 5, 10, 25, and 50 services, which we believe represent most cloud-based applications. Note that today many real-world microservice architectures use a much larger number of microservices. Still, in our experience, the number of microservices with close interactions (like a common compliance rule base) is usually in the range of 5-50. From our point of view, early performance analysis in early architecture design should be focused on such interacting clusters of microservices rather than considering microservices that have little impact on the performance aspects in focus.

Data Set Preparation

We executed each experimental case 5 times and reported minimum, first quartile (Q_1), median, third quartile (Q_3), 95th percentile, maximum, mean, and standard deviation (σ) of the recorded round-trip times. Additionally, a weighted average of median RTT is calculated over the number of cloud services. Let RTT_n be the median round-trip time for a number of cloud services and $WAvg$ the weighted average, which is calculated using the following formula:

$$WAvg = \frac{\frac{RTT_5}{5} + \frac{RTT_{10}}{10} + \frac{RTT_{25}}{25} + \frac{RTT_{50}}{50}}{4} \tag{A.1}$$

The weighted average corresponds to the average RTT per one cloud service in different experiment cases. $WAvg$ is used to normalize the results and make them comparable across the studied architectures.

A.2.3. Statistical Details

This section presents the details of our statistical analysis.

Statistical Methods

We first report descriptive statistics, then perform a multiple regression analysis, a technique used to create prediction models that estimate the value of a dependent variable based on values of two or more independent variables [80]. The following hypotheses are formulated for this experiment:

H_0: There is no significant prediction of the round-trip time RTT of requests by the number of cloud services n_{serv} and call frequencies cf.

H_A: There is a significant prediction of the round-trip time RTT of requests by the number of cloud services n_{serv} and call frequencies cf.

We created four prediction models for each architecture configuration, in total 16 models, to estimate the RTT, as the dependent variable, based on call frequency and the number of cloud services, as the independent variables. Out of these models, we report those two for each architecture configuration with the lowest *p-values* (i.e., those with the highest statistical significance of the predicted results). We used the R language[8] for our statistical analysis.

Cross-Validation of the Regression Models

To cross-validate the regression models and assess their predictive abilities [70], we performed a second run of our experiments with slightly changed parameters. We used our regression models to estimate round-trip times in multiple instances of the number of cloud services and call frequencies. Then, we ran the experiments, recorded RTT values, and compared the actual results with the estimated ones. The number of cloud services has been chosen to differ from those we trained our models with, but they are in the same range, i.e., 30, 40, and 60 services. Furthermore, to find out if our regression models can estimate results for a call frequency not used in the original data set, in addition to 500 and 1000, we used a new call frequency of 800 r/s, for which four desktop computers are utilized to generate the load.

A.3. Analysis

This section presents the analysis of our results.

A.3.1. Experiment Results

Table A.2 presents the experimental results of all architectures for the multi-cloud-node scenario. We can see that for CE, when taking the same number of cloud services, increasing call frequency from 100 to 500 r/s results in a nonlinear rise of median RTT of more than five times. However, when we double the call frequency from 500 to 1000 r/s, the median increases almost linearly. We observe the same trend with a weighted average of round-trip times.

[8] https://www.r-project.org

Table A.2.: Experimental Results of All Architectures

Arch.	cf (r/s)	n_{serv}	Min. RTT (ms)	Q_1 (ms)	Median RTT (ms)	Q_3 (ms)	95th Percentile (ms)	Max. RTT (ms)	Mean RTT (ms)	σ	WAvg (ms)
CE	100	5	25.598	31.377	37.884	77.283	466.155	631.252	98.059	136.076	37.131
		10	62.059	107.9975	136.734	323.3727	1179.303	1292.048	325.288	367.279	
		25	351.298	1191.604	1438.716	1849.310	3131.327	3325.957	1651.817	732.885	
		50	1558.662	3198.889	3486.238	3888.954	8763.811	8813.965	4174.527	1938.576	
	500	5	14.965	951.199	1114.719	1274.4817	1413.479	1998.354	1033.383	336.828	295.815
		10	199.221	2485.889	2669.604	3080.838	3403.013	3959.199	2664.332	546.876	
		25	1848.500	7482.008	8077.590	8519.928	9132.945	9987.151	7843.596	1089.819	
		50	8800.803	17552.022	18510.818	19030.105	19701.113	20501.393	18043.396	1569.938	
	1000	5	10.021	2116.356	2770.525	3229.294	3914.324	4909.937	2673.797	842.366	727.784
		10	213.977	5622.522	6741.707	7574.174	8658.320	15762.526	6484.570	1584.818	
		25	4928.274	16971.840	19620.402	21115.877	22539.220	35547.201	18650.974	3394.748	
		50	12779.84	39582.05	44902.12	47242.38	49364.520	75751.217	42879.598	7349.506	
DR_3	100	5	12.407	17.962	20.614	45.938	219.418	358.606	49.327	67.591	8.979
		10	24.475	31.143	42.555	117.595	622.318	786.069	131.759	184.488	
		25	58.763	110.093	178.123	619.097	1948.941	2145.527	534.422	649.009	
		50	207.219	517.992	1020.674	1554.446	3530.587	3681.506	1381.369	1080.652	
	500	5	9.1600	20.973	54.699	142.444	510.137	663.389	122.964	155.557	111.427
		10	36.760	378.067	497.419	590.276	1146.401	1484.203	506.953	265.814	
		25	57.249	2505.792	3965.498	4866.302	5527.488	6279.350	3676.808	1444.207	
		50	570.627	8326.662	11320.266	12351.667	13032.551	13774.296	10092.286	2917.993	
	1000	5	7.070	329.4105	595.3050	893.1605	1499.987	2576.772	653.093	437.785	241.631
		10	15.125	1185.353	1946.702	2856.923	3497.138	4297.438	1994.626	1035.537	
		25	22.686	4440.487	6943.560	9629.477	10971.649	12175.990	6937.950	2862.820	
		50	4085.752	11233.909	18752.584	24458.039	26567.286	28846.745	17818.120	6870.081	
DR_6	100	5	13.130	18.081	21.482	53.835	292.091	439.211	66.442	92.213	5.345
		10	24.523	31.956	41.547	154.331	588.913	692.254	140.782	187.354	
		25	59.725	88.546	143.640	698.237	1612.439	1669.844	478.032	550.105	
		50	129.424	249.395	359.192	1872.233	3896.844	4033.159	1273.733	1377.067	
	500	5	6.053	56.572	83.383	130.114	456.810	587.507	126.677	123.246	47.934
		10	15.222	133.602	226.448	354.651	695.318	860.288	278.271	194.367	
		25	51.818	883.619	1432.079	1801.626	2371.261	5103.557	1450.629	852.522	
		50	202.680	2709.620	4746.551	6618.730	7636.278	9139.602	4639.890	2152.850	
	1000	5	6.0190	362.8822	563.8395	722.0797	1279.516	1679.144	568.558	326.043	125.817
		10	6.369	575.932	829.978	1170.606	2386.818	3425.810	938.111	614.172	
		25	28.540	2183.593	3450.883	4693.387	5855.607	6720.997	3367.412	1595.319	
		50	71.612	5548.778	8473.436	11046.606	13570.286	15530.022	8232.277	3465.464	
SA	100	5	13.939	18.297	21.818	53.860	174.662	465.173	48.827	96.112	5.218
		10	27.784	34.110	42.278	175.466	601.827	731.133	142.529	188.345	
		25	66.538	93.071	130.398	639.999	1411.885	1570.485	426.281	465.568	
		50	144.760	201.866	353.252	1280.374	2977.373	3112.574	981.886	1031.220	
	500	5	7.019	25.249	49.632	93.276	308.843	462.864	80.743	89.948	14.494
		10	16.027	99.895	142.365	183.956	1523.994	1685.055	308.145	448.920	
		25	44.503	316.710	426.120	771.040	1702.167	2350.455	625.794	490.145	
		50	162.088	658.954	838.437	1052.560	1882.903	2297.493	893.149	431.288	
	1000	5	5.155	169.817	346.633	547.109	1148.907	1695.961	412.302	328.463	55.265
		10	3.111	385.790	674.881	1259.548	2086.450	3670.457	852.858	643.786	
		25	12.249	864.723	1152.719	1539.998	2848.677	4187.7180	1320.540	761.473	
		50	45.632	1341.721	1906.893	2643.657	3641.671	4805.411	2011.770	942.661	

standard deviations are highest in CE compared to the other architectures. This is explainable since there is only one service that receives all requests and checks compliance rules, i.e., the central entity service. At the beginning of a run, lower round-trip times are captured; however, as more requests arrive and this service becomes overloaded, delays become larger, resulting in higher RTT towards the end of a run. This causes the high σ in the CE architecture.

For DR_3, as expected, we achieve a lower mean RTT compared to CE since we have three routers that can process requests simultaneously instead of one central entity that processes requests in sequence. Similarly, the weighted averages are reduced compared to CE. We can see that choosing a higher number of cloud services, i.e., 25 and 50, results in an almost linear rise of median round-trip times when increasing call frequency from 500 to 1000 r/s. We observe lower standard deviations for DR_3 compared to CE, most likely because three routers become less overloaded than only one central entity service.

For DR_6 and SA, when having five or ten cloud services, we see almost identical numbers. This is because, in our implementation, which aims to implement the architectures in a comparable way, these architectures are identical when having 5 cloud services and only slightly different when having ten. With the increase of cloud services to 25 and 50, we also increase the number of sidecars in SA but still have only six routers in DR_6, resulting in higher numbers in median round-trip times, weighted averages, and standard deviations in DR_6 compared to SA. In both of these architecture configurations, we can see an almost linear increase of median round-trip times when having 25 and 50 cloud services and doubling call frequency from 500 to 1000 r/s.

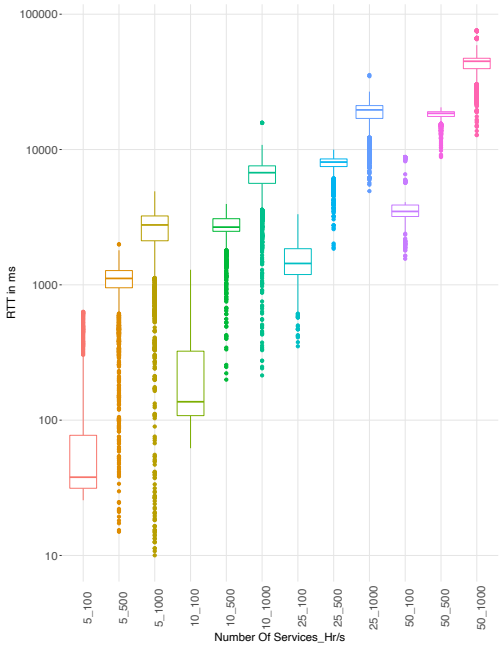
Furthermore, in SA, we observe an almost linear rise of median RTT when we increase the number of cloud services but keep the call frequency constant. SA results in lower standard deviations compared to the other architectures in most cases, likely because we have more controlling services, i.e., sidecars, which process incoming requests simultaneously; therefore, they become less overloaded.

Figures A.1 and A.2 show the distribution of RTT values for each experimental case for all architecture configurations. It can be observed that round-trip times rise when the number of cloud services or call frequencies increase. Another obvious observation is the decrease of σ when moving from CE to DR and SA architectures. In CE, we observe a rather low interquartile range. By adding more controlling services, i.e., dynamic routers and sidecars, we get a higher interquartile range in DR and SA.

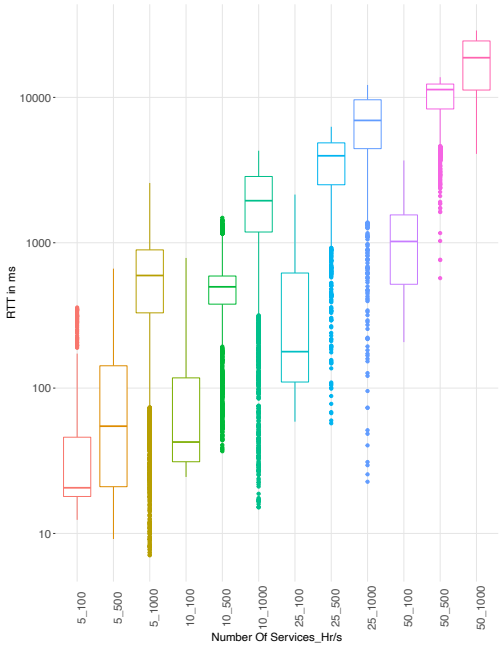
An interesting observation is that in all architectures, the outliers mostly lie between minimum RTT and Q_1 except for the call frequency of 100 r/s. As explained before, at the beginning of a run, round-trip times are very low, and as more requests arrive, the times go up. In the case of 100 r/s, since the frequency of calls are not so high that they can overload cloud nodes, the majority of the RTT values stay in the lower range, and only some calls are delayed, resulting in outliers plotted above the interquartile range.

A.3.2. Multiple Regression Analysis

Table A.3 presents the top two prediction models based on statistical significance for each kind of architecture that we created based on our multiple regression analysis. All of our

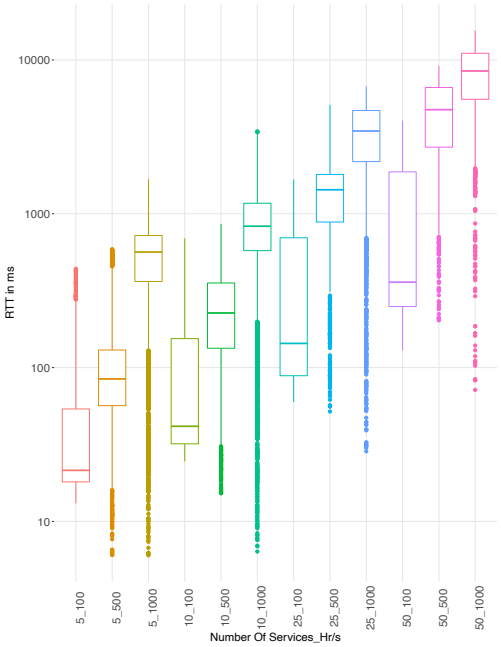


(a) Central Entity

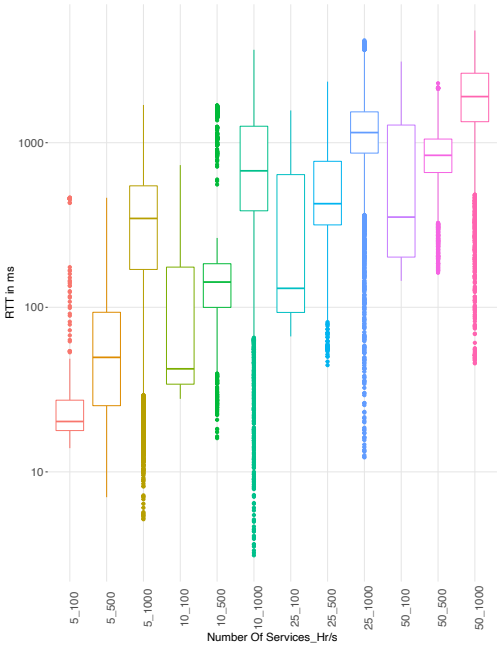


(b) Dynamic Routers_3

Figure A.1.: Distribution of Experimental Results for CE and DR_3 Architectures



(a) Dynamic Routers_6



(b) Sidecar-Based Architecture

Figure A.2.: Distribution of Experiment Results for DR_3 and SA Architectures

Table A.3.: Prediction Models

Arch.	*SC:* Service Coefficient	*FC:* Frequency Coefficient	*IC:* Interaction Coefficient	*Int:* Intercept	F-statistic: p-value
CE	-82.506e+00	-2.503e+00	0.974e+00	22.994e+00	<2.2e-16
	6.693e+02	1.677e+00	-	-5.196e+03	<2.2e-16
DR_3	3.844e+01	-1.124e+00	3.488e-01	-7.251e+02	<2.2e-16
	3.242e+02	6.724e+00	-	-7.156e+03	<2.2e-16
DR_6	2.298e+01	-2.006e-01	1.516e-01	-4.635e+02	<2.2e-16
	1.472e+02	3.211e+00	-	-3.259e+03	<2.2e-16
SA	6.987e+00	5.215e-01	2.573e-02	-1.290e+02	<2.2e-16
	28.072e+00	1.100e+00	-	-603.403e+00	<2.2e-16

models result in a very low *p-value* which allows us to reject the null hypothesis and accept the alternative hypothesis that the number of cloud services and call frequency affect the RTT. For cross-validation and assessment of the predictive abilities of the models, we ran our experiments for a second time and compared the results with predictions of all models. We found that using interaction terms in our regression models results in more accurately predicted round-trip times. The interaction term tells us that the effect of n_{serv} on the estimated RTT changes with different values of cf (and vice versa). Let SC be the service coefficient, FC the frequency coefficient, IC the interaction coefficient, and Int the intercept. Our prediction models are based on the following formula:

$$RTT = SC \cdot n_{serv} \quad + \quad FC \cdot cf \quad + \quad IC \cdot n_{serv} \cdot cf \quad + \quad Int \tag{A.2}$$

Table A.4 compares predicted round-trip times with the second run of our empirical results. We ordered the reported statistics based on n_{serv} as it makes comparison easier. Except for the central routing, all predicted round-trip times lie in the interquartile range of the actual results. In the case of CE, although our prediction models have a very low *p-value* suggesting the statistical significance of the predicted results, some estimated round-trip times are slightly higher than Q_3 or lower than Q_1. However, the predictions are not far off. This is explainable since there are large standard deviations in CE, as shown in Table A.2. Therefore, prediction models are trained on a widely-spread set of data. In the case of DR and SA, we get lower standard deviations meaning the round-trip times are not as widely spread as in the case of CE. We can observe that our predictions are very close to the median of the actual results with DR_3 and DR_6 under call frequencies of 800 and 1000 r/s.

A.3.3. Single-Node Environment

In addition to our main focus on multi-cloud-node performance, we studied single-node performance to get a rough comparison of the impact of distributed communications. Therefore, we executed the experiments on a single-node setup and compared the results (only in the overview) to the multi-node configuration.

Weighted averages of all architectures can be seen in Figure A.3. The $WAvg$ is calculated when implemented on single-node or multi-cloud-node environments. The figure illustrates

Table A.4.: Second Run of Empirical Results and the Predicted Round-Trip Times

Arch.	n_{serv}	cf (r/s)	$\mathbf{Q_1}$ (ms)	**Median** ***RTT*** (ms)	$\mathbf{Q_3}$ (ms)	**Estimated** ***RTT*** (ms)
CE	**30**	**500**	9083.668	10119.824	10459.467	10909.028
		800	14888.795	16157.245	17029.664	18925.731
		1000	18815.264	22402.055	24335.038	24270.199
	40	**500**	14001.234	14620.704	14911.775	14954.882
		800	20937.26	24084.266	25230.38	25894.125
		1000	29397.73	33857.476	35722.81	33186.954
	60	**500**	19996.99	21326.758	22240.68	19000.736
		800	33497.17	37169.810	38182.12	39830.913
		1000	45520.67	52991.573	56520.75	51020.461
DR_3	**30**	**500**	3379.516	5842.062	6577.077	5098.099
		800	5301.085	7948.498	9864.858	7900.100
		1000	5920.507	9535.263	12997.476	9768.100
	40	**500**	6274.714	8476.561	9322.539	7226.500
		800	8282.451	10954.531	13713.971	11074.900
		1000	9102.496	13627.311	19351.622	13640.500
	60	**500**	10534.364	14531.970	15798.849	11483.300
		800	10876.653	17206.1335	21745.091	17424.500
		1000	13964.748	22423.751	30804.183	21385.300
DR_6	**30**	**500**	13964.748	2224.4155	30804.183	2399.600
		800	2313.973	3310.6625	4531.752	3703.819
		1000	3022.064	4394.1155	6576.130	4573.300
	40	**500**	2038.993	3654.794	4973.425	3387.400
		800	2994.578	5081.697	6194.052	5146.420
		1000	3894.233	6426.936	8054.121	6319.100
	60	**500**	3454.181	5145.119	7964.733	5363.000
		800	5533.973	8065.514	10947.300	8031.620
		1000	6101.927	9551.363	13187.443	9810.700
SA	**30**	**500**	551.958	691.218	719.194	727.310
		800	825.058	1017.261	1135.050	1115.33
		1000	958.909	1353.016	1681.329	1374.01
	40	**500**	657.204	894.185	923.209	925.830
		800	974.586	1278.070	1458.266	1391.040
		1000	1313.515	1713.703	2148.084	1701.180
	60	**500**	851.668	1180.670	1486.216	1322.870
		800	1213.473	1608.923	2069.630	1942.46
		1000	1606.438	2172.414	2690.043	2355.520

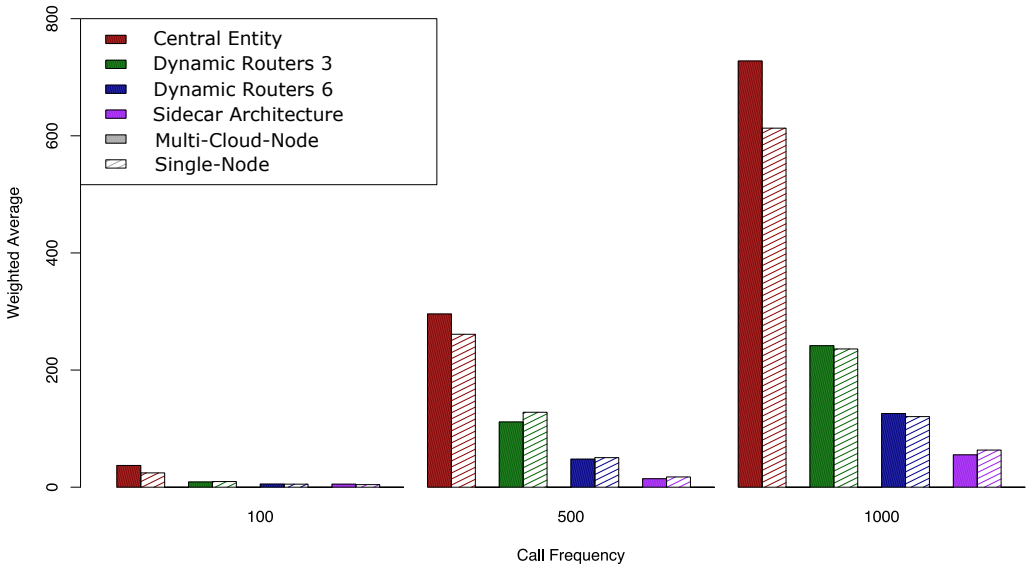


Figure A.3.: Weighted Averages of Architecture Configurations on Single-Node and Multi-Cloud-Node Environments

that CE always performs poorly compared to other architectures. Moreover, CE has even worse performance when implemented on a multi-node environment compared to the single-node setup (a higher weighted average means lower performance). This is expected since all cloud services must send requests to the central entity service. Implementing CE on multiple cloud nodes requires additional network communications between different hosts.

The performance differences of the other three architectures are negligible when deployed on single- or multi-node configurations under 100 r/s. However, at 500 r/s, we can see that DR_3 and DR_6 perform better on multiple nodes. With 1000 r/s, they perform slightly better in a single-node environment. SA results in performance increases on the higher frequencies of 500 and 1000 r/s when implemented on multi-cloud-node compared to a single-node setup. These results suggest that adding more cloud resources, i.e., controlling services and cloud nodes, does not always result in a significant performance improvement. It can be seen that increasing the number of controlling services from 1 in CE to 3 in DR_3 and doubling it to 6 in DR_6 enhances performance considerably. However, having 1 sidecar per each service results in a marginal boost in performance. The same is true when adding more cloud nodes. Although we see the highest performance of all architecture configurations in the case of SA on a multi-cloud-node environment, the difference when implemented on a single node is marginal.

A.4. Threats to Validity

A threat to internal validity refers to the extent to which independent variables caused the observed variation in the dependent variables. Concerning this threat, we ensured that all experiment artifacts were deployed on the same infrastructure with the same distribution of cloud services. Furthermore, the experiment applications are composed using the exact same cloud service instance so that there are no implementation differences between the services that might affect the dependent variables. Nonetheless, such internal validity threats cannot be completely excluded. In particular, despite our careful implementation and deployment work, some aspects may have been slightly different in the multiple implementations and deployments. We mitigated this threat by carefully double-checking all technical aspects of our experiment. We ensured the machines we ran our study on were idle, but possibly other services, e.g., of the operating systems, may have influenced our measurements. We mitigated this threat through a warm-up phase and multiple experiment runs.

One external threat to validity is that potentially our experiment setup for cloud environments is not chosen well. As a result, the setup might not be comparable to real-world scenarios hosted on cloud infrastructures. Another related threat is that we implemented all three architecture options from scratch in Node.js. We did not use existing implementations of these options to make the implementations comparable in an experiment. However, the threat remains that our implementations might not represent the existing off-the-shelf tools like Envoy[7] for sidecars or enterprise service buses for central entities. These threats are at odds with internal validity. We modeled, implemented, and similarly deployed the tested architectures as much as possible to ensure comparability. From our experience, they are close to existing architectures in the cloud. However, further research is needed to transfer our results to specific technologies and settings. The cloud services are deployed using container technology Docker[3], which is commonly used in cloud-based architectures. Real-world cloud applications are often composed of different computing and storage services offered by multiple cloud providers. Such scenarios may have additional effects on the performance of the evaluated architectures.

A.5. Conclusions

In this appendix, we investigated three representative service- and cloud-based architectures for making and enacting dynamic data-flow-routing decisions concerning their performance. To do so, we designed an experiment and used our findings as follows. Firstly, we created prediction models providing estimation for architects on the performance impact of the investigated architectures. The found models show high statistical significance. Moreover, we cross-validated the estimated round-trip times with measurements from an additional experiment run. Our results show that using more controlling services improves the performance as expected (i.e., the more distributed settings show better performance, especially under high load). However, there appears to be a cut-off point, after which the gained performance increase is marginal. The architect must then

decide, in relation to other relevant qualities of the viable architectural options, whether the performance is of the highest priority or a trade-off is acceptable.

Secondly, we calculated *RTT* weighted averages of all configurations. We did so to compare the performance of architectures when services are implemented locally or distributed on multiple cloud nodes. While a single-node setup leads to a noticeable improvement of the central entity architecture in our settings, the performance differences in the other three architecture configurations are less evident. In those cases, the negative impact of distributed communication and the positive impact of more cloud resources almost balance out in most of our empirical results.

B. Trade-Offs Adaptation: Statistical Performance Model

This appendix presents our approach to automatically adapt reliability and performance trade-offs. We study a statistical performance model to address the research problem P_3: *Lack of an approach to automatically adapt the reliability and performance trade-offs*. Our work is published as a conference paper [14] and is the basis of the dissertation. Chapter 5 studies the approach presented here further and uses our analytical performance model, which is generalizable to other infrastructures.

B.1. Introduction

Many dynamic routing architectures are available, including sidecar-based routing, routing through a central entity such as an event store or gateway, or architectures with multiple routers. These architectures are based on vastly different implementation concepts, such as API gateways [79], enterprise service buses [26], message brokers [45], or sidecars [61, 49, 36]. However, they essentially all dynamically route or block incoming requests. We propose the details of our approach, which abstracts these architectural patterns using one Adaptive Dynamic Routers (ADR) architecture. We hypothesize that a dynamic self-adaptation of the routing architecture is beneficial over any fixed architecture selections for reliability and performance trade-offs. That is, if encountered with traffic and load changes, our approach dynamically self-adapts between more central or distributed routing to optimize system reliability and performance. This adaptation is automated based on a multi-criteria optimization analysis [4]. We evaluate our approach by analyzing our empirical data during an experiment of 1200 hours of runtime (see Section 2.3 for details). Our extensive systematic evaluation of 1089 cases confirms that our hypothesis holds and our approach is beneficial regarding reliability and performance. Even on average, our novel architecture offers, on average, 14.0% higher reliability gain compared to completely distributed routing and 7.5% more performance gain compared to centralized routing. Moreover, we empirically validate our results on Google Cloud Platform[1] infrastructure.

The structure of this appendix is as follows: In Section B.2, we explain the proposed adaptive dynamic routers architecture in detail, and in Section B.3 provide the parameterization of our models. Section B.4 presents the evaluation of the presented approach. Section B.5 discusses the threats to the validity of our research. Finally, we conclude in Section B.6.

[1] `https://cloud.google.com`

Table B.1.: The Mathematical Notations Used in this Appendix

Notation	Description
R	Reliability model
$R_{service}$	Reliability model for service crashes
R_{router}	Reliability model for router crashes
P	Performance model
T	Observed system time
CI	Crash interval
cf	Incoming call frequency
Com	Set of all components
CP_c	Crash probability of a component c every CI
d_c	Expected average downtime after a component c crashes
IR_T	Total number of requests exchanged between components
n_c^{exec}	Number of successfully executed requests before the crash of a component c
n_{serv}	Number of services in an ADR instance
n_{rout}	Number of routers in an ADR instance
$s_{crashed}$	A service s when crashed
$r_{crashed}$	A router r when crashed
A	Allocation of routers
Sc	Service coefficient
Rc	Router coefficient
Fc	Frequency coefficient
SFc	Service vs. frequency coefficient
RFc	Router vs. frequency coefficient
SRc	Service vs. router coefficient
$SRFc$	Service vs. router vs. frequency coefficient
$R_{n_{rout}}$	Reliability of an ADR architecture configuration by the number of routers
$P_{n_{rout}}$	Performance of an ADR architecture configuration by the number of routers
R_{th}	Reliability threshold
P_{th}	Performance threshold
$MAPE$	Mean absolute percentage error
$ErrR$	Prediction error regarding reliability
$ErrP$	Prediction error regarding performance
$\overline{\Delta R}$	Reliability average percentage difference
$\overline{\Delta P}$	Performance average percentage difference
$RGain$	Reliability gain
$PGain$	Performance gain
$RWeight$	Reliability weight
$PWeight$	Performance weight
Q_1	First quartile
Q_3	Third quartile
σ	Standard deviation
$\overline{\Delta R}$	Performance average percentage difference
$\overline{\Delta P}$	Reliability average percentage difference
R_c	Reliability of an evaluation case c
P_c	Performance of an evaluation case c
$model_c$	Result of the model for experimental case c
$empirical_c$	Measured empirical data for experimental case c
$Cases$	Set of all incoming call frequencies and the number of services
n_c	Length of $Cases$

B.2. Approach Details

We introduce the ADR reliability and performance models and present our reconfiguration algorithm. Table B.1 presents the mathematical notation used in this appendix.

B.2.1. ADR Models

This section summarizes our reliability model reported in Chapter 3. Moreover, we present a statistical performance model, which is based on the approach presented in Chapter 4.

Reliability Model

We used Bernoulli processes [90] to model request loss during router and service crashes. Request loss was defined as the number of incoming requests that were not processed due to a failure, such as a crash of a component. We calculated the total request loss during an observed system time T as a metric of reliability:

$$R = \lfloor \frac{T}{CI} \rfloor \cdot cf \cdot \sum_{c \in Com} CP_c \cdot d_c \cdot (IR_T - n_c^{exec}) \tag{B.1}$$

Remember that CI is the crash interval, i.e., the interval in which we check for a crash of a component. Assume the Heartbeat pattern [46] is used to check the system health, CI is the time between two consecutive health checks. cf is the incoming call frequency based on requests per second (r/s), C is the set of components, i.e., routers and services, CP_c is the crash probability of each component, d_c is the average downtime of a component after it crashes, IR_T is the number of requests exchanged between components, and n_c^{exec} is the number of successfully executed requests before the crash of a component. Note that IR_T and n_c^{exec} must be parameterized based on the application presented in the next section.

To empirically validate our model, we ran an extensive experiment (see Section 2.3 for details). We compared our analytical model of reliability with the empirical results of our experiment by using the Mean Absolute Percentage Error (MAPE) [90]. With more experiment runs, we observed an ever-decreasing error, converging at 8.1%. We also double-checked the accuracy of our models with three other error metrics, i.e., Mean Absolute Error (MAE), Mean Square Error (MSE), and Root Mean Square Error (RMSE), which yielded the same trend of the prediction error being constantly reduced.

Performance Model

Remember that RTT is the round-trip time. During our experiment, we recorded RTT of each request, which was defined as the difference in time from the moment a request was received until it was routed through all cloud services involved in the processing of the request. The round-trip times indicated the performance impact of the studied architectures, i.e., the Central Entity (CE), Dynamic Routers (DR), and the Sidecar-based Architecture (SA) presented in Section 2.1.

ADR is a reconfigurable architecture that can be configured as any of the aforementioned architectures by changing the router configurations. We introduce a variable n_{rout}, which defines the number of routers in an ADR instance. To illustrate, CE has one router that processes requests centrally. Therefore, ADR with $n_{rout} = 1$ indicates the central-entity architecture. On the other hand, the SA architecture is completely distributed and has one router, i.e., a sidecar, per service. The number of routers in SA is the same as the number of services (n_{serv}). Consequently, the sidecar-based architecture is ADR with $n_{rout} = n_{serv}$. The dynamic routers architecture was a middle ground with three routers, i.e., $n_{rout} = 3$, among which we distributed services equally.

For the study in this appendix, we did a multiple regression analysis [80] on the recorded round-trip times. We created a prediction model of performance for ADR based on the independent variables n_{rout}, n_{serv}, and cf. Note that the nonlinear regression in Equation (B.2) is system-specific and needs to be performed for each application separately. We used the R language[2].

$$\begin{aligned} P = & Int + (Sc \cdot n_{serv}) + (Rc \cdot n_{rout}) + (Fc \cdot cf) + \\ & (SFc \cdot n_{serv} \cdot cf) + (RFc \cdot n_{rout} \cdot cf) + \\ & (SRc \cdot n_{serv} \cdot n_{rout}) + (SRFc \cdot n_{serv} \cdot n_{rout} \cdot cf) \end{aligned} \tag{B.2}$$

Table B.2 reports the coefficients of our performance model. As it can be seen, all of the calculated coefficients of our regression model resulted in a very low *p-value* that indicates a high statistical significance of the prediction results.

As before, we used the MAPE [90] to compare the ADR performance prediction model with the recorded round-trip times. The prediction error converged at 14.7% with more experiment runs. We double-checked the error with MAE, MSE, and RMSE. Note that the regression analysis needs to be performed for each application separately. Since in the cloud performance field, 30.0% is commonly used as the target prediction accuracy [59], and the fact that our focus is to have a rough prediction of the impact of performance when architecting a system, the prediction accuracy is reasonable.

Table B.2.: Performance Prediction Model

Coefficient (Related Variable)	**Sc** (n_{serv})	**Rc** (n_{rout})	**Fc** (cf)	**SFc** ($n_{serv} : cf$)	**RFc** ($n_{rout} : cf$)	**SRc** ($n_{serv} : n_{rout}$)	**SRFc** ($n_{serv} : n_{rout} : cf$)	**Int** (*Intercept*)
F-statistic p-value	<2.2e-16							
Value	4.128 e+00	-3.032 e+00	-2.249 e-01	3.682 e-02	1.228 e-02	1.308 e-01	-4.530 e-03	1.714 e+01

B.2.2. ADR Reconfiguration Algorithm

This section presents the details of our reconfiguration algorithm.

[2] https://www.r-project.org

Multi-Criteria Optimization Analysis

In our approach, the reconfiguration between the architecture configurations is performed automatically based on a Multi-Criteria Optimization (MCO) analysis [4]. Consider the following optimization problem: An application using the ADR architecture has n_{serv} services and is under stress for a period of time with the call frequency of cf. To optimize reliability and performance, the system can change between different ADR architecture configurations dynamically by adjusting the number of routers (n_{rout}), ranging from the extreme of a centralized routing (only one router), over any dynamic router configurations, up to the extreme of one router per service (n_{serv} routers), i.e., the SA configuration.

We use the notations $R_{n_{rout}}$ and $P_{n_{rout}}$ to specify the reliability and performance of the respective architecture configurations by their number of routers. For instance, only configuring one router R_1 indicates the reliability model of an ADR with only one router (i.e., the CE architecture), and configuring n_{serv} routers (i.e., $R_1, \ldots, R_{n_{serv}}$) indicates SA. R_{th} and P_{th} are the reliability and performance thresholds.

$$
\begin{aligned}
&Minimize \\
&\quad R_{n_{rout}} && \text{(B.3)} \\
&\quad P_{n_{rout}} && \text{(B.4)} \\
&Subject\ to \\
&\quad R_{n_{rout}} \leq R_{th} && \text{(B.5)} \\
&\quad P_{n_{rout}} \leq P_{th} && \text{(B.6)} \\
&\quad 1 \leq n_{rout} \leq n_{serv} && \text{(B.7)}
\end{aligned}
$$

The MCO question is: Given a cf and n_{serv}, what is the optimal number of routers which minimizes request loss and average RTT per each request without the predicted ADR reliability and performance going beyond a certain threshold? Typically, there is no single answer to an MCO problem. Using the above MCO analysis, we find a range of n_{rout} configurations that all meet the constraints. One end of this range optimizes reliability and the other performance, thus, we need a preference function so ADR can automatically select an n_{rout} value. Appendix B.3.3 presents an illustrative example.

Preference Function

We can define different criteria to choose the final ADR router configuration. We study a scenario: In Chapter 3, we empirically validated that centralized routing offers higher reliability than distributed approaches. On the other hand, Chapter 4 showed that decentralized routing improves performance by processing incoming requests in parallel. An architect can define operational profiles for low and high levels of cf, based on which ADR adapts to more centralized routing (a lower n_{rout} to improve reliability) or more distributed approaches (a higher n_{rout} to improve performance). As shown in Figures 2.5 and 2.6, the Quality of Service (QoS) monitor observes the incoming call frequency and triggers the manager to reconfigure the routers when there is a change in these defined cf levels.

Based on these operational profiles, the preference function instructs ADR to choose a final n_{rout} value in the range found by the MCO analysis. This selection is based on an importance vector that gives weights to reliability and performance given by Algorithm 9. For example, when reliability is of the highest importance to an application, an architect gives the highest weight, i.e., 1.0, to reliability and the lowest weight, i.e., 0.0, to performance. Thus, the preference function chooses the lowest value on the n_{rout} range to choose more centralized routing. This selection results in higher reliability, as illustrated in Appendix B.3.3.

Automatic Reconfiguration

Figure 2.5 shows that the QoS monitor reads the monitoring data from the API gateway. The monitor feeds this information to the manager that deploys new routers or reconfigures the existing ones. Algorithm 9 presents our reconfiguration algorithm used by the scheduler. The reconfiguration algorithm is triggered, for instance, whenever reliability or performance metrics degrade, e.g., observed as an increase in request loss or a decrease of average RTT. Time intervals, changes in incoming load, or a different route with more or fewer services can also trigger the algorithm if more appropriate than metrics degradation (see Figure 2.6 for details).

Algorithm 9: Reconfiguration Algorithm for the Adaptation of Reliability and Performance Trade-Offs

```
Input: R_th, P_th, performanceWeight

R_nrout, P_nrout, cf, n_serv ← consumeRPData()
routersRange ← MCO(cf, n_serv, R_nrout, P_nrout, R_th, P_th)
reconfigSolution ← preferenceFunction(routersRange,
                                      performanceWeight)
reconfigureRouters(reconfigSolution)

function preferenceFunction(range, PW)
begin
    length ← max(range) - min(range) +1
    floor ← ⌊ PW * length ⌋
    if floor == max(range) then
        return max(range)
    else if floor == 0 then
        return min(range)
    else
        return floor + min(range) -1
    end
end
```

`reconfigureRouters(reconfigSolution)` in Algorithm 9 performs the final reconfiguration based on the chosen solution. This step differs based on an ADR application and needs to be specified. In this appendix, we assume a very simple and cost-ineffective strategy of starting the new configuration in parallel with the running setup. Afterward, when the infrastructure is ready to process the incoming requests, we change to the new configuration and tear down the old setup. Note that the purpose of this appendix is to provide a scientific proof-of-concept and not a fully functioning code base. The simple strategy fits this purpose and does not result in reliability or performance degradation because of the reconfiguration.

B.3. Parameterization of Model to Experiment Parameter Values

As mentioned, our analytical reliability model is general for routing architectures. Therefore, it needs to be parameterized based on the application. Note that the performance model is a regression analysis that has to be performed for each application separately. We parameterize Equation (B.1) for our experiment by specifying request loss for the ADR with the introduction of the n_{rout} variable, i.e., the number of routers.

B.3.1. Experiment Details

This section presents the details of our experiment (see Section 2.3 for general details).

Experiment on Private Cloud

We ran an experiment of 200 runs with a total of 1200 hours of runtime (excluding setup time). We had a private cloud setting with three physical nodes, each having two identical Intel® Xeon® E5-2680 CPUs. We installed Virtual Machines (VMs) with eight cores and 60 GB of system memory. Each router or service was containerized in a Docker[3] container. We deployed one router exclusively on one VM for CE, three routers each deployed on one VM for DR, and one router per service deployed on the same VM for SA.

We had three levels for the number of services (n_{serv}), i.e., 3, 5, and 10 services, and four levels for incoming call frequency (cf), i.e., 10, 25, 50, and 100 r/s, totaling twelve experimental cases for each of the three architecture (36 cases overall). We utilized five desktop computers for load generation, each hosting an Intel® Core™ i3-2120T CPU with 8 GB of system memory, which used Apache JMeter[4] to send Hypertext Transfer Protocol (HTTP) version 1.1[5] requests to the VMs. The routing of requests was as follows: For the sake of simplicity, we labeled the services incrementally from 1 and let the incoming requests go through all services one by one. An example of an experiment configuration is shown in Figure B.1.

[3] https://www.docker.com
[4] https://jmeter.apache.org
[5] https://tools.ietf.org/html/rfc7230

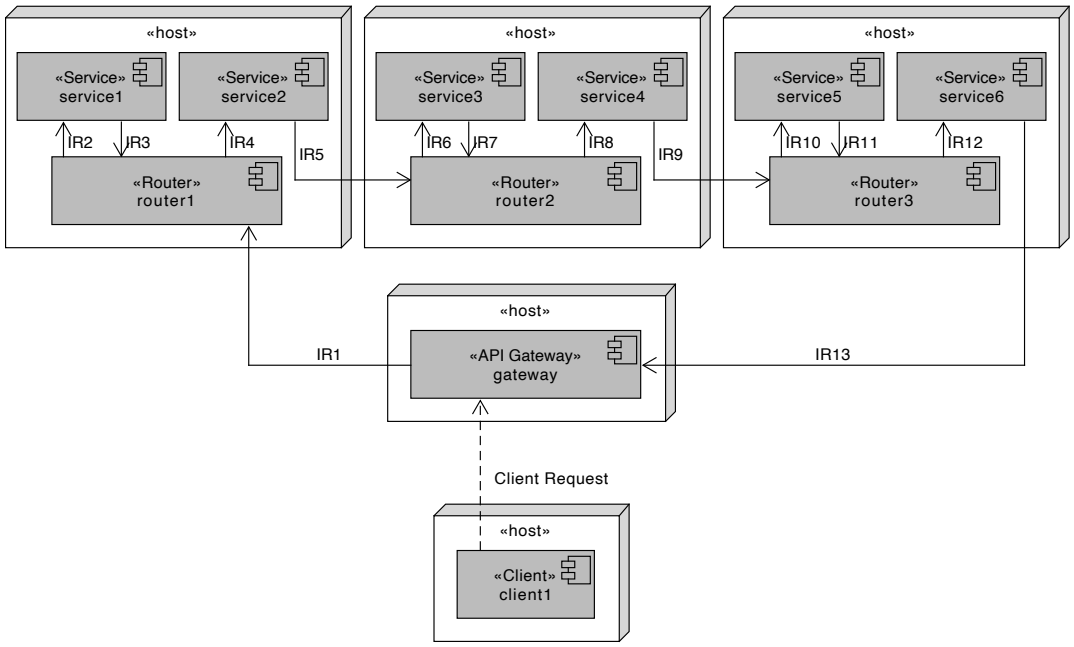


Figure B.1.: Example ADR/Experiment Configuration

Validation Experiment on Public and Private Clouds

We use our private cloud to have control over the infrastructure and have repeatable experiment runs. On a public cloud, other factors can influence the results, such as the parallel workload of other applications or the physical distance of the node. To show that our approach can be used on other infrastructures as well, we empirically validate the analysis of an illustrative sample case (see Appendix B.3.3) once on our private cloud infrastructure and once on Google Cloud Platform (GCP)[1]. We run our experiment a second time using the values of the sample case, i.e., with 10 services, i.e., $n_{serv} = 10$ and three call frequencies, i.e., 25, 50, and 100 r/s. In this scenario, ADR changes the routing configuration, i.e., the value of n_{rout}, automatically. On GCP, we started E2 machine instances[6] with 2 vCPUs and 8 GB of memory and duplicated our experiment infrastructure.

B.3.2. Parameterization

Figure B.1 shows an example ADR (and experiment) configuration with three routers and six services. When a router or a service crashes, some requests are not processed. To know how many of these requests are lost, we use the total number of requests (IR_T), from which we subtract the number of already executed ones (n_c^{exec}). Let us consider an

[6] https://cloud.google.com/compute/docs/general-purpose-machines

example: Assume *service*5 crashes in the example configuration. In this case, $IR1$ to $IR9$ are processed ($n_c^{exec} = 9$) but $IR10$ to $IR13$ (four requests) are lost. In this example, we can see that there are 13 requests ($IR_T = 13$). Therefore, the number of lost requests in this example is:

$$IR_T - n_c^{exec} = 13 - 9 = 4 \tag{B.8}$$

We calculate IR_T based on the number of services as the following since there are one incoming and one outgoing request per each service and one request from the gateway:

$$IR_T = 2n_{serv} + 1 \tag{B.9}$$

In the example ADR configuration, we have $n_{serv} = 6$, so $IR_T = 13$. To calculate n_c^{exec}, we must differentiate between service and router crashes. We define $s_{crashed}$ as the label number of the crashed service. In our experiment for **service crashes**, we have:

$$n_c^{exec} = 2s_{crashed} - 1 \tag{B.10}$$

Note that in our example case, we considered the crash of *service*5, i.e., $s_{crashed} = 5$, and calculated $n_c^{exec} = 9$. We observed the system for 10 minutes (600 seconds) for each experimental case and checked for a crash every 15 seconds with a uniform crash probability of 0.5% for all components (see Section 2.3.4 for experiment cases):

$$T = 600\,s \tag{B.11}$$

$$CI = 15\,s \tag{B.12}$$

$$CP_c = 0.5\% \tag{B.13}$$

Therefore, we can rewrite Equation (B.1) for service crashes using Equations (B.9) to (B.13):

$$R_{service} = 0.6 \cdot cf \cdot n_{serv}(n_{serv} + 1) \tag{B.14}$$

In case of a router crash, we define the allocation of routers (A) as a set indicating the number of directly linked services of each router. For instance, the allocation of routers in the example ADR configuration (see Figure B.1) is:

$$A = \{2, 2, 2\} \tag{B.15}$$

In our experiment, services were equally allocated to routers:

$$A = \{\frac{n_{serv}}{n_{rout}}, \frac{n_{serv}}{n_{rout}}, ..., (\frac{n_{serv}}{n_{rout}} \pm 1)\} \tag{B.16}$$

in which A has the length of n_{rout}. In Figure B.1, there are six services, i.e., $n_{serv} = 6$, and three routers, i.e., $n_{rout} = 3$. Therefore, we have the allocation presented in Equation (B.15).

Let $r_{crashed}$ be the label number of the crashed router. For **router crashes**, we have:

$$n_c^{exec} = 2 \sum_{r=1}^{r_{crashed}} A_{r-1} \tag{B.17}$$

Therefore, we can rewrite Equation (B.1) for router crashes as:

$$R_{router} = 0.6 \cdot cf \cdot [n_{serv} + n_{rout}(n_{serv} + 1)] \tag{B.18}$$

Finally, we can rewrite Equation (B.1) by adding Equations (B.14) and (B.18) as:

$$R = 0.6 \cdot cf \cdot [(n_{serv})^2 + (n_{rout} + 2)n_{serv} + n_{rout}] \tag{B.19}$$

B.3.3. Illustrative Sample Case

We provide an illustrative example to explain our concepts.

Multi-Criteria Optimization

We customize the MCO analysis presented in Appendix B.2.2 with our experiment models' setup, using Equations (B.2) and (B.19):

$$\begin{aligned}
&Minimize \\
&R_{n_{rout}} = 0.6 \cdot cf \cdot [(n_{serv})^2 + (n_{rout} + 2)n_{serv} + n_{rout}] && \text{(B.20)} \\
&P_{n_{rout}} = 17.14+ \\
&\quad 4.128 \cdot n_{serv} - 3.032 \cdot n_{rout} - 0.2249 \cdot cf+ \\
&\quad 0.1308 \cdot n_{serv} \cdot n_{rout} + 0.03682 \cdot n_{serv} \cdot cf+ \\
&\quad 0.01228 \cdot n_{rout} \cdot cf - 0.00453 \cdot n_{serv} \cdot n_{rout} \cdot cf && \text{(B.21)} \\
&Subject\ to \\
&\quad R_{n_{rout}} \leq R_{th} && \text{(B.22)} \\
&\quad P_{n_{rout}} \leq P_{th} && \text{(B.23)} \\
&\quad 1 \leq n_{rout} \leq n_{serv} && \text{(B.24)}
\end{aligned}$$

Operational Profiles

Let us consider an example ADR application having ten services, i.e., $n_{serv} = 10$, which is operational for the expected input call frequency of $10 \leq cf \leq 100$ r/s with a reliability threshold of $R_{th} = 10000$ request loss per 10 minutes of the experiment (on average a very high rate of 16.667 requests per second), and a performance threshold of $P_{th} = 60\ ms$ average RTT per each request. We do the MCO analysis for different chunks of call

frequency starting with the lower bound, i.e., $cf = 10$ r/s:

$$\begin{aligned} &\textit{Minimize} \\ &\quad R_{n_{rout}} = 720 + 66 \cdot n_{rout} && \text{(B.25)} \\ &\quad P_{n_{rout}} = 59.853 - 2.0542 \cdot n_{rout} && \text{(B.26)} \\ &\textit{Subject to} \\ &\quad R_{n_{rout}} \leq 10000 && \text{(B.27)} \\ &\quad P_{n_{rout}} \leq 60\ ms && \text{(B.28)} \\ &\quad 1 \leq n_{rout} \leq 10 && \text{(B.29)} \end{aligned}$$

In Equations (B.25) and (B.26), the performance and reliability thresholds are always satisfied in the range of $1 \leq n_{rout} \leq 10$. In Appendix B.2.2, we mentioned an example preference function with an importance weight of 1.0 for reliability and 0.0 for performance giving the highest priority to reliability. This preference function chooses the lowest possible value for n_{rout}, i.e., it favors more centralized routing to improve reliability. Using the example function, we can choose $n_{rout} = 1$, i.e., a central routing configuration, on the lower bound of the expected frequency range ($cf = 10\ r/s$). We now find the highest possible cf where central routing is still applicable, in other words, the reliability and performance predictions are below the thresholds. Using Equations (B.20) and (B.21) when $n_{rout} = 1$ and $n_{serv} = 10$:

$$\begin{aligned} R_1 &= 78.6 \cdot cf \leq 10000 && \text{(B.30)} \\ P_1 &= 56.696 + 0.11028 \cdot cf \leq 60\ ms && \text{(B.31)} \end{aligned}$$

Within the expected frequency range, i.e., $10 \leq cf \leq\ r/s$, the reliability predictions are always below the threshold. However, when solving the performance model given by Equation (B.31), the highest acceptable frequency is $cf = 29.960\ r/s$, with which central routing stays within the defined thresholds. Remember in Appendix B.2.2, we mentioned that an architect could define operational profiles for the incoming frequency. Therefore, we can define a low level for cf, in which central routing is reasonable as:

$$10.000 \leq cf \leq 29.960\ r/s \qquad \text{(B.32)}$$

We take the higher bound of the operational profile in the above formula, i.e., $cf =$ 29.960 r/s, and repeat the process in Equations (B.25) and (B.31). Table B.3 reports all operational profiles of the expected incoming frequency, i.e., $10 \leq cf \leq 100$, and

Table B.3.: Operational Profiles of Incoming Call Frequency for the Illustrative Example

n_{serv}	cf **Operational Profile** (r/s)	n_{rout}
10	$10.000 \leq cf \leq 29.960$	1
	$29.961 \leq cf \leq 65.079$	2
	$65.080 \leq cf \leq 100.00$	3

the respective final reconfiguration choice based on the preference function. Note that as soon as there is a call frequency outside of this range, the QoS monitor triggers the manager to reconfigure the routers (see Figure 2.5). We consider a frequency level from our experiment in each operational profile, i.e., $cf \in \{25, 50, 100\}$ r/s. Table B.4 presents the ADR model predictions.

Table B.4.: ADR Reliability and Performance Predictions for the Illustrative Example based on its Operational Profiles

n_{serv}	cf (r/s)	n_{rout}	R	P (ms)
10	**25**	**1**	1965.000	59.453
	50	**2**	4260.000	58.835
	100	**3**	9180.000	57.672

B.3.4. Empirical Validation

Table B.5 presents the empirical measurements of the adaptation using the ADR architecture on our private cloud and the GCP infrastructure. We calculate the prediction error of our models reported in Table B.4 using the Mean Absolute Percentage Error (MAPE) [90]. Let $model_c$ and $empirical_c$ be the result of the model and the measured empirical data for the experiment case c, $Cases$ the set of all incoming call frequencies and the number of services, and n_c the length of $Cases$:

$$MAPE = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \left| \frac{model_c - empirical_c}{empirical_c} \right| \tag{B.33}$$

The prediction errors of our models are reported in Table B.5. Let $ErrR$ and $ErrP$ be the prediction error regarding reliability and performance, respectively. On our experiment infrastructure, we have $ErrR = 8.9\%$ and $ErrP = 17.0\%$. As mentioned, the performance model is a system-specific regression analysis that needs to be performed for each application separately. Therefore, on GCP, we only empirically validate the reliability model. The prediction error regarding reliability on GCP is $ErrR = 14.0\%$. Given the commonly used target prediction accuracy of 30.0% in the cloud quality of service research [59], these results are reasonable.

Table B.5.: ADR Empirical Measurements for the Illustrative Example Case

n_{serv}	cf (r/s)	n_{rout}	R	P (ms)	$ErrR$ (%)	$ErrP$ (%)	R	$ErrR$ (%)
			Experiment Infrastructure				Google Cloud Platform	
10	**25**	**1**	2450.000	54.116	8.852	16.973	1792.000	14.015
	50	**2**	4434.000	44.100			6014.000	
	100	**3**	9448.000	53.577			9486.000	

B.4. Evaluation

In this section, we evaluate the ADR architecture by comparing the ADR performance and reliability predictions to the empirical results of our experiment. The ADR architecture is not specific to our experiment infrastructure or cases. Architects can freely use the proposed architecture and adjust it to their needs, as explained in the last section. That is, we use our empirical data set in the online artifact of this thesis[6] to evaluate ADR using measured data of an extensive experiment.

Systematic Analysis

In the last section, we studied one sample case with specific reliability and performance thresholds, plus an example preference function. To systematically evaluate our proposed ADR architecture, we go through a range of thresholds and importance weights for reliability and performance. We compare ADR model predictions (see Table B.4 for an example) with our experiment cases reported in Appendix B.3.1. That is, we compare ADR with its fixed counterparts, i.e., CE ($n_{rout} = 1$), DR ($n_{rout} = 3$), and SA ($n_{rout} = n_{serv}$). As mentioned, we had three levels for n_{serv}, i.e., 3, 5, and 10, for each of which we consider the expected incoming call frequency of $10 \leq cf \leq 100\ r/s$ (see Appendix B.3.3 for an illustrative example). In this range, we studied four levels of cf, i.e., 10, 25, 50, and 100 r/s, in our experiment. Therefore, we have nine experiment cases: Three architectures, each configured with three n_{serv} values, which are operational for four levels of call frequencies.

Regarding reliability and performance thresholds, we start with very tight reliability and very loose performance thresholds so that only centralized routing is acceptable. We slightly increase the reliability and decrease the performance thresholds by 10% in each step so that distributed routing becomes applicable. To find the starting points, we take the worst-case scenario of our empirical data into consideration. In Equation (B.19), a higher n_{serv} results in a higher expected request loss. In our experiment cases, the highest value for n_{serv} is 10 services. As mentioned before, CE is the most reliable, and SA gives the best performance. With $n_{serv} = 10$, the worst-case reliability and performance predictions for CE are 7860 request losses per 10 minutes of system time, i.e., 13.1 requests per second and 67.724 ms average RTT per each request. On the other hand, the worst-case predictions for SA with ten services are 13800 requests per 10 minutes of experiment, i.e., 23.0 requests per second and 39.311 ms average RTT. We adjust these values slightly and take our boundary thresholds as follows:

$$8000 \leq R_{th} \leq 14000 \tag{B.34}$$

$$40 \leq P_{th} \leq 70\ ms \tag{B.35}$$

We start with an importance weight of 1.0 for reliability and 0.0 for performance giving the highest priority to reliability. We decrease the reliability importance (consequently increasing the performance weight) by 10% in each iteration. We evaluate 1089 systematic evaluation cases: 11 threshold and 11 importance weight levels, each evaluated for 9

Table B.6.: Statistics of the Gain Percentages

	Reliability			Performance		
Arch.	**CE**	**DR**	**SA**	**CE**	**DR**	**SA**
Min	-27.923	-18.245	-2.344	-2.240	-11.390	-22.848
$\boldsymbol{Q_1}$	-5.854	-2.1640	-0.658	-0.324	-8.798	-11.809
Median	-1.845	4.762	6.125	1.845	-4.762	-6.125
$\boldsymbol{Q_3}$	0.324	8.798	11.810	5.854	2.164	0.658
Max	2.240	11.390	22.848	27.923	18.245	2.344
$\boldsymbol{\sigma}$	6.204	7.000	7.607	6.204	7.000	7.607
Mean	-3.981	2.535	7.089	3.981	-2.535	-7.089
Mean RWeight > 0.5	-0.963	8.145	13.997	0.963	-8.145	-13.997
Mean RWeight < 0.5	-7.454	-3.550	0.073	7.454	3.550	-0.073

experimental cases. To support reproducibility, the evaluation script and the log containing detailed information on each systematic evaluation case are available in the online artifact of this dissertation[6]. Let $\overline{\Delta R}$ and $\overline{\Delta P}$ be the average percentage difference of reliability and performance, and R_c and P_c the reliability and performance of the evaluation case c. Remember that $R_{n_{rout}}$ and $P_{n_{rout}}$ are the reliability and performance predictions of an ADR architecture configuration by the number of routers. We calculate the average percentage differences as follows:

$$\overline{\Delta R} = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{R_c - R_{n_{rout}}}{R_{n_{rout}}} \tag{B.36}$$

$$\overline{\Delta P} = \frac{100\%}{n_c} \cdot \sum_{c \in Cases} \frac{P_c - P_{n_{rout}}}{P_{n_{rout}}} \tag{B.37}$$

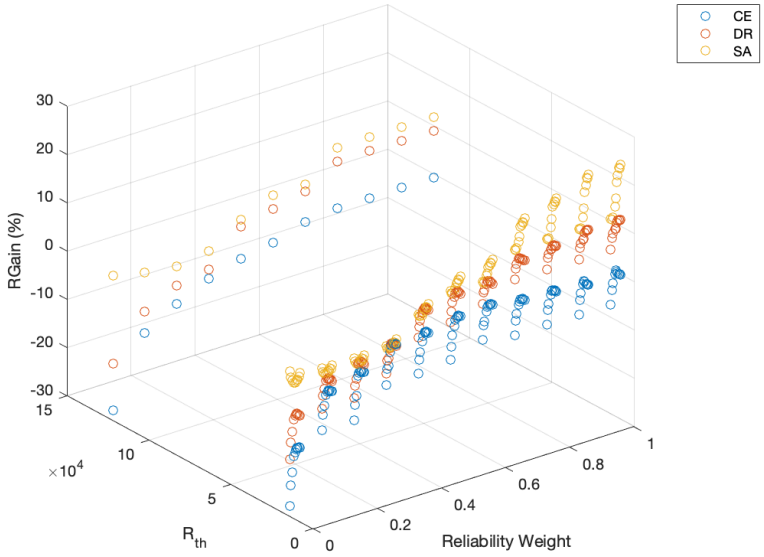
Here, $Cases$ is the set of all incoming call frequencies and the number of services, i.e., $cf \in \{10, 25, 50, 100\}$ and $n_{serv} \in \{3, 5, 10\}$. Therefore, $n_c = 12$, i.e., the length of $Cases$.

We define reliability gain as the percentage of reliability improvement in contrast to the performance worsening. Similarly, performance gain is defined as the percentage of performance increase compared to reliability degradation:

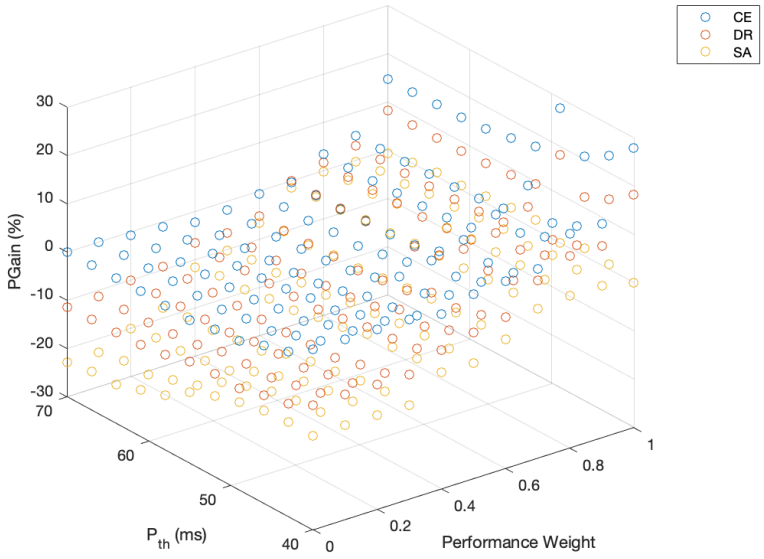
$$RGain = \left|\overline{\Delta R}\right| - \left|\overline{\Delta P}\right| \tag{B.38}$$

$$PGain = -RGain \tag{B.39}$$

Figure B.2 shows the reliability and performance gains of ADR compared to the CE, DR, and SA architectures. As seen in Figure B.2a, ADR almost always has the highest reliability gain compared to SA in our experiment cases. This is expected because SA is a completely distributed routing extreme resulting in the lowest reliability compared to the other architectures. Note that a higher number of routers results in a higher request loss,



(a) Reliability Gain



(b) Performance Gain

Figure B.2.: ADR Reliability and Performance Gains Compared to CE, DR and SA Architectures

according to Equation (B.19). As the reliability weight increases and the performance importance lowers, the reliability gain improves.

The dynamic routers, i.e., a static configuration with three routers, were considered a middle ground of the three studied architectures in our experiment. Here, the same holds true, i.e., the reliability and performance gains of DR are between those of CE and SA architectures. Figure B.2b shows that ADR has the highest performance gain compared to CE. However, the ADR performance gain does not differ greatly from the architectures. We investigate the statistics of the data reported in Table B.6, in which Q_1, Q_3, σ, and $RWeight$ are the first and the third quartiles, the standard deviation and the reliability weight, respectively. Note that according to Equation (B.39), the statistics regarding the mean of performance gains are the negative of those of reliability gains as shown in Table B.6. The mean of data is calculated over 121 cases, i.e., 11 threshold levels and 11 weights.

This investigation should illustrate that in cases where the wrong architecture choice is made, significant increases in reliability or performance are offered by ADR, i.e., 22.8% reliability gain compared to SA and 27.9% performance gain compared to CE architecture. Let us now investigate the mean of data, i.e., where cases with correct and incorrect architecture choices are analyzed together, to show that even here, ADR provides improvements. The mean performance gain compared to CE is 4.0%, averaged over all cases, ADR gains more performance than it loses reliability compared to the centralized routing. When taking those cases into account, in which performance has a higher importance weight than reliability, i.e., $RWeight < 0.5$, the mean percentage gain for CE is 7.5%. On the other hand, the mean reliability gain for SA is 7.1%, i.e., ADR offers a higher reliability gain than performance loss compared to a completely distributed routing averaged over all experiment cases. Taking only those cases where reliability is of higher importance than performance, i.e., $RWeight > 0.5$, the mean reliability gain is 14.0%.

B.5. Threats to Validity

In this section, we discuss the threats to the validity, along with some limitations of our study.

B.5.1. Construct Validity

We used request loss and the round-trip times of requests as metrics of reliability and performance, respectively. While this is a common approach in service- and cloud-based research (see Section 1.3), the threat remains that other metrics might model these quality attributes better, e.g., a cascade of calls beyond a single call sequence for reliability [67], or data transfer rates of messages which are m byte-long for performance [53]. More research, probably with real-world systems, is required for this threat to be excluded.

B.5.2. Internal Validity

The dynamic-routing architectures are based on many different technologies. Our ADR architecture abstracts the controlling logic component in dynamic routing under a concept called router to allow interoperability between these architectures. In a real-world system, changing between these technologies is not an easy task, but it is not impossible either. In this appendix, we provided a scientific proof-of-concept based on an experiment with the prototypical implementation of these technologies. The threat remains that, in a real-world application, changing between these technologies might have other impacts on reliability and performance, e.g., network latency increasing processing time.

Moreover, we considered a simple reconfiguration strategy to start the new setup in parallel with the running configuration to avoid impacts on reliability, e.g., request loss due to reconfiguration and performance, e.g., increased processing time while reconfiguring. In a real-world system, this solution is cost-ineffective that introduces additional resource demands. The architects must specify a reconfiguration strategy based on their application needs to mitigate this threat.

B.5.3. External Validity

We designed our novel architecture with generality in mind and explained in detail how architects could specify ADR to their needs (see Appendix B.3). In spite of the fact that we systematically evaluated ADR using the data of our extensive experiment of 1200 hours with 1089 evaluation cases, the threat remains that evaluating ADR based on another infrastructure may lead to different results. To mitigate this thread, we empirically validated our measurements on the Google Cloud Platform[1] infrastructure and showed that our results are applicable.

B.6. Conclusions

We proposed the details of the Adaptive Dynamic Routers (ADR) architecture that automatically adjusts performance and reliability trade-offs based on a multi-criteria optimization analysis. We systematically evaluated ADR using 1089 evaluation cases based on the empirical data of our extensive experiment of 1200 hours of runtime (see Appendix B.3.1 for details). Our results show that ADR can adapt the architecture configuration in a running system to optimize reliability and performance. If the wrong architectural choice has been made, ADR can lead to substantial gains in reliability or performance.

Even on average, where cases with the right and the wrong architecture choice are analyzed together, ADR offers good results. For example, on average, it offers 14.0% higher reliability than performance loss compared to completely distributed routing when reliability is of higher importance, and 7.5% more performance gains than reliability decrease compared to centralized routing when performance is a higher priority. Moreover, we empirically validated our model predictions on our private experiment infrastructure

and the GCP public cloud. The empirical validation had a prediction error of 14.0%, which indicates that our approach is applicable outside of our private infrastructure.

To the best of our knowledge, there has not been any architecture presented in the literature that automatically adjusts reliability and performance trade-offs, specifically in service- and cloud-based dynamic routing. Our proposed ADR architecture adapts, based on triggers, e.g., change of incoming load frequency or degradation of monitoring data, to an optimal configuration to prevent request loss or increase of round-trip times. Prior to our work, architects needed to manually redesign and redeploy architecture configurations.

Bibliography

[1] S. Ahamad and Ratneshwer. Some studies on performability analysis of safety critical systems. *Computer Science Review*, 39:100319, 2021.

[2] S. P. Ahuja and A. Patel. Enterprise service bus: A performance evaluation. *Communications and Network*, 3(03):133, 2011.

[3] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya. Archeopterix: An extendable tool for architecture optimization of AADL models. In *ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software, MOMPES 2009*, pages 61–71. IEEE, 2009.

[4] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Trans. Software Eng.*, 39(5):658–683, 2013.

[5] A. Amiri, C. Krieger, U. Zdun, and F. Leymann. Dynamic data routing decisions for compliant data handling in service- and cloud-based architectures: A performance analysis. In *IEEE International Conference on Services Computing (SCC)*, 2019.

[6] A. Amiri, E. Ntentos, U. Zdun, and S. Geiger. Tool support for learning architectural guidance models and pattern mining from architectural design decision models. In *European Conference on Pattern Languages of Programs (EuroPLoP)*, forthcoming.

[7] A. Amiri and U. Zdun. Cost-aware multifaceted reconfiguration of service- and cloud-based dynamic routing applications. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2023.

[8] A. Amiri and U. Zdun. Smart and adaptive routing architecture: An internet-of-things traffic manager based on artificial neural networks. In *IEEE International Conference on Software Services Engineering (SSE)*, 2023.

[9] A. Amiri and U. Zdun. Tool support for the adaptation of quality of service trade-offs in service- and cloud-based dynamic routing architectures. In *European Conference on Software Architecture (ECSA)*, forthcoming.

[10] A. Amiri, U. Zdun, and K. Plakidas. Stateful depletion and scheduling of containers on cloud nodes for efficient resource usage. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2022.

[11] A. Amiri, U. Zdun, G. Simhandl, and A. van Hoorn. Impact of service- and cloud-based dynamic routing architectures on system reliability. In *International Conference on Service Oriented Computing (ICSOC)*, 2020.

[12] A. Amiri, U. Zdun, and A. van Hoorn. Modeling and empirical validation of reliability and performance trade-offs of dynamic routing in service- and cloud-based architectures. In *IEEE Transactions on Services Computing (TSC)*, 2021.

[13] A. Amiri, U. Zdun, and A. van Hoorn. Analytical modeling and empirical validation of performability of service- and cloud-based dynamic routing architecture patterns. In *IEEE Transactions on Services Computing (TSC)*, forthcoming.

[14] A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Automatic adaptation of reliability and performance tradeoffs in service- and cloud-based dynamic routing architectures. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2021.

[15] A. Amiri, U. Zdun, A. van Hoorn, and S. Dustdar. Cost-aware multidimensional auto-scaling of service- and cloud-based dynamic routing to prevent system overload. In *IEEE International Conference on Web Services (ICWS)*, 2022.

[16] P. Arcaini, E. Riccobene, and P. Scandurra. Modeling and analyzing mape-k feedback loops for self-adaptation. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 13–23. IEEE, 2015.

[17] P. Arcaini, E. Riccobene, and P. Scandurra. Formal design and verification of self-adaptive systems with decentralized control. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(4):1–35, 2017.

[18] R. Bankston and J. Guo. Performance of container network technologies in cloud environments. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*, pages 0277–0283. IEEE, 2018.

[19] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[20] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):800–813, 2018.

[21] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, page 54–65, New York, NY, USA, 2007. ACM.

[22] M. Beltrán. Automatic provisioning of multi-tier applications in cloud computing environments. *The Journal of Supercomputing*, 71:2221—2250, 2015.

[23] S. Bock and M. Weiß. A proof of local convergence for the adam optimizer. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[24] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Architecture-based reliability prediction with the palladio component model. *IEEE Transactions on Software Engineering*, 38(6):1319–1339, 2011.

[25] A. Busch, D. Fuchss, and A. Koziolek. Peropteryx: Automated improvement of software architectures. In *IEEE International Conference on Software Architecture ICSA Companion 2019*, pages 162–165. IEEE, 2019.

[26] D. A. Chappell. *Enterprise service bus*. O'Reilly, 2004.

[27] R. C. Cheung. A user-oriented software reliability model. *IEEE transactions on Software Engineering*, pages 118–125, 1980.

[28] A. Chung, J. W. Park, and G. R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, 2018.

[29] J. Comden, S. Yao, N. Chen, H. Xing, and Z. Liu. Online optimization in cloud resource provisioning: Predictions, regrets, and algorithms. In *Publication: Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2019.

[30] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-based software performance analysis*. Springer, 2011.

[31] C. Czepa, A. Amiri, E. Ntentos, and U. Zdun. Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability. *Software and Systems Modeling*, pages 3331–3371, 2019.

[32] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah. A new docker swarm scheduling strategy. In *2017 IEEE 7th international symposium on cloud and service computing (SC2)*, 2017.

[33] D. J. Dean, H. Nguyen, P. Wang, and X. Gu. Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[34] M. Di Mauro, G. Galatro, M. Longo, F. Postiglione, and M. Tambasco. Performability analysis of containerized ims through queueing networks and stochastic models. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8, 2022.

[35] D. Dossot, J. D'Emic, and V. Romero. *Mule in action*. Manning Greenwich, 2014.

[36] Envoy. Service mesh. *https://www.learnenvoy.io/articles/service-mesh.html*, 2019.

[37] G. Galante and L. C. E. de Bona. A survey on cloud computing elasticity. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE, 2012.

[38] V. Grassi and S. Patella. Reliability prediction for service-oriented computing environments. *IEEE Internet Computing*, 10(3):43–49, 2006.

[39] G. Grimmett and D. Welsh. *Probability: An Introduction*. Cambridge University Press, 1986.

[40] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Strohmeier A. (eds) Reliable Software Technologies - Ada-Europe '96*. Springer, 1996.

[41] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger. Performance engineering for microservices: research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 223–226. ACM, 2017.

[42] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 23–27, 2013.

[43] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda. Bungee: An elasticity benchmark for self-adaptive iaas cloud environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 46–56, Piscataway, NJ, USA, 2015. IEEE Press.

[44] A. R. Hevner, S. T. March, and S. Ram. Design science in information systems research. In *MIS Q.*, volume 28(1), pages 75–105, 2004.

[45] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[46] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson. *Cloud Design Patterns*. Microsoft Press, 2014.

[47] D. G. D. L. Iglesia and D. Weyns. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):1–31, 2015.

[48] K. Indrasiri. *Beginning WSO2 ESB*. Apress, 2016.

[49] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[50] C. Kaewkasi and K. Chuenmuneewong. Improvement of container scheduling for docker using ant colony optimization. In *IEEE 10th International Conference on Autonomic Computing*, 2013.

[51] V. V. Kalashnikov. *Mathematical Methods in Queuing Theory*. Springer, 2013.

[52] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu. Efficiency analysis of provisioning microservices. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pages 261–268. IEEE, 2016.

[53] N. Kratzke. About microservices, containers and their underestimated impact on network performance. *arXiv preprint arXiv:1710.04049*, 2017.

[54] G. Kumar, M. Kaushik, and R. Purohit. Reliability analysis of software with three types of errors and imperfect debugging using markov model. *International Journal of Computer Applications in Technology (IJCAT)*, 2018.

[55] A. Lisnianski, E. Levit, and L. Teper. Short-term availability and performability analysis for a large-scale multi-state system based on robotic sensors. *Reliability Engineering and System Safety*, 205:107206, 2021.

[56] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, and V. Chang. A new container scheduling algorithm based on multi-objective optimization. In *Soft Computing, 22(23), 7741-7752*, 2018.

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

[58] M. A. Matalytski. Analysis and forecasting of expected incomes in markov networks with bounded waiting time for the claims. *Automation and Remote Control*, 76(6):1005–1017, 2015.

[59] D. A. Menascé and V. A. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, 2001.

[60] T. Menouer. Kcss: Kubernetes container scheduling strategy. In *The Journal of Supercomputing, 77(5), 4267-4293*, 2021.

[61] Microsoft. Sidecar pattern. `https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar`, 2010.

[62] Y. Mo, L. Xing, L. Zhang, and S. Cai. Performability analysis of multi-state sliding window systems. *Reliability Engineering and System Safety*, 202:107003, 2020.

[63] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

[64] G. Montavon, W. Samek, and K.-R. Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, 2018.

[65] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):44, 2016.

[66] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing*, 2013.

[67] M. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[68] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. In *IEEE Transactions on Software Engineering*. IEEE, 2019.

[69] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[70] R. R. Picard and R. D. Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, 1984.

[71] R. Pietrantuono, S. Russo, and A. Guerriero. Run-time reliability estimation of microservice architectures. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–35. IEEE, 2018.

[72] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske. Hora: Architecture-aware online failure prediction. *Journal of Systems and Software*, 137, 2017.

[73] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske. An architecture-aware approach to hierarchical online failure prediction. In *12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, 2016.

[74] T. Rademakers and J. Dirksen. *Open-Source ESBs in Action: Example Implementations in Mule and ServiceMix*. Simon and Schuster, 2008.

[75] A. Rago, S. A. Vidal, J. A. Diaz-Pace, S. Frank, and A. van Hoorn. Distributed quality-attribute optimization of software architectures. In *Proceedings of the 11th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2017*, pages 7:1–7:10. ACM, 2017.

[76] P. Raj, A. Raman, and H. Subramanian. *Architectural Patterns: Uncover essential patterns in the most indispensable realm*. Packt Publishing, December 2017.

[77] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach*. The MIT Press, 2016.

[78] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.

[79] C. Richardson. Microservice architecture patterns and best practices. *http://microservices.io/index.html*, 2019.

[80] D. L. Rubinfeld. Reference guide on multiple regression. *Federal Judicial Center, 2nd edition*, 2000.

[81] V. S. Sharma and K. S. Trivedi. Architecture based analysis of performance, reliability and security of software systems. In *Proceedings of the 5th International Workshop on Software and Performance*, WOSP '05, page 217–227, New York, NY, USA, 2005. Association for Computing Machinery.

[82] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *2nd ACM Symposium on Cloud Computing*, 2011.

[83] T. Shezi, E. Jembere, and M. Adigun. Performance evaluation of enterprise service buses towards support of service orchestration. In *Proc. of International Conference on Computer Engineering and Network Security (ICCENS'2012)*, 2012.

[84] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *37th International Conference on Software Engineering (ICSE)*, 2015.

[85] R. Smith, K. Trivedi, and A. Ramesh. Performability analysis: measures, an algorithm, and a case study. *IEEE Transactions on Computers*, 37(4):406–417, 1988.

[86] B. Spitznagel and D. Garlan. Architecture-based performance analysis. In *Proc. the 1998 Conference on Software Engineering and Knowledge Engineering*. Carnegie Mellon University, June 1998.

[87] O. Sukwong, A. Sangpetch, and H. S. Kim. Sageshift: managing slas for highly consolidated cloud. In *2012 Proceedings IEEE INFOCOM*, pages 208–216, 2012.

[88] M. Sureshkumar and P. Rajesh. Optimizing the docker container usage based on load scheduling. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, 2017.

[89] M. Torquato, P. Maciel, and M. Vieira. Model-based performability and dependability evaluation of a system with vm migration as rejuvenation in the presence of bursty workloads. *Journal of Network and Systems Management*, 30(1):3, 2021.

[90] K. S. Trivedi and A. Bobbio. *Reliability and availability engineering: modeling, analysis, and applications*. Oxford University Press, 2017.

[91] V. K. Vaishnavi and K. W. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. Auerbach, 2007.

[92] A. Van Hoorn, A. Aleti, T. F. Düllmann, and T. Pitakrat. Orcas: Efficient resilience benchmarking of microservice architectures. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 146–147. IEEE, 2018.

[93] K. Vandikas and V. Tsiatsis. Performance evaluation of an iot platform. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 141–146. IEEE, 2014.

[94] L. Wang, X. Bai, L. Zhou, and Y. Chen. A hierarchical reliability model of service-based software system. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 199–208, July 2009.

[95] R. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[96] C. Wohlin, P. Runeson, M. Hoest, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering*. Springer, 2012.

[97] F. Zhang, X. Tang, X. Li, S. U. Khan, and Z. Li. Quantifying cloud elasticity with container-based autoscaling. *Future Generation Computer Systems*, 98:672–681, 2019.

[98] Z. Zheng and M. R. Lyu. Collaborative reliability prediction of service-oriented systems. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 35–44, May 2010.

Name	**Amirali Amiri, M.Sc.**
Address	Währinger Gürtel 146 / 17, 1090 Vienna
Telephone	+4368110689758
Email	amirali.amiri@univie.ac.at
Webpage	http://cs.univie.ac.at/amirali.amiri
Birth	March 26th, 1989 – Iran
Languages	Persian (Native), English (C2), German (C1)

Education

Now	**Doctoral Candidate in Computer Science**
2017 Apr	University of Vienna, Austria Design and modeling of architectural requirements of cloud-based systems
2015 Sep	**Master of Science in Informatics**
2012 Apr	Technical University of Munich, Germany Computer architecture
2012 Feb	**Bachelor of Science in Computer Engineering**
2007 Sep	Ferdowsi University of Mashhad, Iran Hardware design

Work Experiences

Now	**University of Vienna, Austria**
2017 Apr	Research and teaching staff (Software Architecture Group) Modeling and empirical validation of architectural design decisions
Now	**Siemens Österreich, Austria**
2022 Feb	Machine-learning operations for distributed cyber-physical systems Research project in collaboration with Uni Wien Software Architecture Group
2022 Feb	**fiskaly GmbH, Austria**
2020 Nov	Generic eFiscalization in the cloud for different organizations Research project in collaboration with Uni Wien Software Architecture Group

2016	**open ideas GmbH, Germany**	
Nov – Dec	Embedded software design of an industrial cyber-physical control system	
2015	**University of Texas at Arlington Research Institute, USA**	
Jun – Sep	Cooperative control of cyber-physical systems, e.g., robot cars and drones	
	open ideas GmbH, Germany	
Jan – May	MS Thesis: Evaluation of face recognition algorithms on embedded platforms	
2014	**open ideas GmbH, Germany**	
Aug – Dec	Embedded software design of an industrial cyber-physical control system	
	Ferdowsi University of Mashhad, Iran	
May – Aug	Evaluation of temperature-sensor implementations on Xilinx virtex-5 FPGA	
2013	**Infineon Technologies AG, Germany**	
Jun – Dec	Support related to test of an ARM microprocessor-based system	
	Technical University of Munich, Germany	
May – Jun	Implementation of computer vision applications with OpenCV	
2012	**Technical University of Munich, Germany**	
Aug – Dec	Adaption of a NoC router to enable configuration on Xilinx Virtex-5 NetFPGA	

Teaching

Since 2019 I have taught Master- and Bachelor-level courses in Advanced Software Engineering and Software Engineering 2 as a Senior Universitätsassistent Praedoc.

My responsibilities include scheduling the courses, managing room reservations, setting up Moodle course pages, setting up student Git repositories, forming student teams, supervising students, coordinating tutors, dealing with students (de)registrations, designing semester projects, writing and correcting exams, dealing with student reviews, publishing grades, and supporting three new teachers.

Publications

I have published 11 research papers in top venues, including the A*-ranked TSC journal and A-ranked conferences, e.g., ICSOC. The publication list follows.

List of Publications

- Cost-Aware Multifaceted Reconfiguration of Service- and Cloud-Based Dynamic Routing Applications

  Amirali Amiri, Uwe Zdun

- *IEEE International Conference on Cloud Computing (CLOUD), 2-8 Jul 2023, Chicago, Illinois USA (2023)*

  DOI 10.5281/zenodo.7919227

- Smart and Adaptive Routing Architecture: An Internet-of-Things Traffic Manager Based on Artificial Neural Networks

  Amirali Amiri, Uwe Zdun
  IEEE International Conference on Software Services Engineering (SSE), 2-8 Jul 2023, Chicago, Illinois USA (2023)

  DOI 10.5281/zenodo.7919351

- Cost-Aware Multidimensional Auto-Scaling of Service- and Cloud-Based Dynamic Routing to Prevent System Overload

  Amirali Amiri, Uwe Zdun, André van Hoorn, Dustdar Schahram

  IEEE International Conference on Web Services (ICWS), 11-17 Jul 2022, Barcelona, Spain (2022)

  DOI 10.1109/ICWS55610.2022.00063

- [Stateful Depletion and Scheduling of Containers on Cloud Nodes for Efficient Resource Usage]

  Amirali Amiri, Uwe Zdun, Konstantinos Plakidas

  IEEE International Conference on Software Quality, Reliability and Security (QRS), 5-9 Dec 2022, Quangzhou, China (2022)

  DOI 10.1109/QRS57517.2022.00056

- [Automatic Adaptation of Reliability and Performance Trade-Offs in Service- and Cloud-Based Dynamic Routing Architectures]

  Amirali Amiri, Uwe Zdun, André van Hoorn, Schahram Dustdar

  IEEE International Conference on Software Quality, Reliability and Security (QRS), 6-10 Dec 2021, Hainan Island, China (2021)

  DOI 10.1109/QRS54544.2021.00055

- [Modeling and Empirical Validation of Reliability and Performance Trade-Offs of Dynamic Routing in Service- and Cloud-Based Architectures]

  Amirali Amiri, Uwe Zdun, André van Hoorn

  IEEE Transactions on Services Computing (TSC), ISSN 1939-1374 (2021)

  DOI 10.1109/TSC.2021.3098178

- [Impact of Service- and Cloud-Based Dynamic Routing Architectures on System Reliability]

  Amirali Amiri, Uwe Zdun, Georg Simhandl, André van Hoorn

  International Conference on Service-Oriented Computing (ICSOC), 14-17 Dec 2020, Dubai, UAE (2020)

  DOI 10.1007/978-3-030-65310-1_13

- [Dynamic Data Routing Decisions for Compliant Data Handling in Service- and Cloud-Based Architectures: A Performance Analysis]

  Amirali Amiri, Christoph Krieger, Uwe Zdun, Frank Leymann

  IEEE International Conference on Services Computing (SCC), 8-13 Jul 2019, Milan, Italy (2019)

  DOI 10.1109/SCC.2019.00044

- [Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability]

  Christoph Czepa, Amirali Amiri, Evangelos Ntentos, Uwe Zdun

  Software and Systems Modeling, 18 pp. 3331-3371 ISSN 1619-1366 Springer (2019)

  DOI 10.1007/s10270-019-00721-4

- [Performance evaluation metrics for ring-oscillator-based temperature sensors on FPGAs: A quality factor]

  Navid Rahmanikia, Amirali Amiri, Hamid Noori, Farhad Mehdipour

  INTEGRATION, the VLSI journal, 57 pp. 81-100 ISSN 0167-9260 (2017)

  DOI 10.1016/j.vlsi.2016.12.007

- [Exploring Efficiency of Ring Oscillator-Based Temperature Sensor Networks on FPGAs (Abstract Only)]

  Navid Rahmanikia, Amirali Amiri, Hamid Noori, Farhad Mehdipour

  Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 22-24 Feb 2015, Monterey, California, USA (2015)

  DOI 10.1145/2684746.2689104