Improving the Performance and Availability of Microservice-Based Cloud Applications

by

Alireza Goli

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering

University of Alberta

Abstract

With the advent of cloud computing, many organizations are moving their software systems to cloud environments. This migration from on-premises to cloud infrastructure has led to the emergence of cloud applications. One of the essential concepts around cloud applications and cloud services is Quality of Service (QoS). Cloud providers are responsible for maintaining the QoS of the cloud services according to the Service Level Agreement (SLA). In this thesis, we focus on performance as one of QoS's dimensions and explore three different ways to improve the performance of cloud applications.

In the first part of this thesis, we use the serverless paradigm to improve the performance of a cloud application with a monolithic architecture running on a single virtual machine. The results suggest that the serverless paradigm improves the performance of the application under study by 93x.

In the second part of this thesis, we introduce an autoscaling approach for microservice applications that leverages machine learning to model the performance behaviour of each service in the application and the performance impact of services on each other. The results show that the proposed autoscaling method improves the performance of microservice applications compared to Kubernetes HPA, the de facto standard for autoscaling in the industry.

In the third part of this thesis, we leverage software multi-versioning as a cost-effective way to improve the performance and availability of cloud applications, particularly in situations where our computational resources are limited. The results show that multi-versioning can be used as an effective

method for maintaining the performance of cloud applications at the desired level.

Preface

This thesis is mainly based on three research works which have been conducted in the Performant and Available Computing Systems (PACS) Lab led by Dr. Hamzeh Khazaei and Sustainable Computing Lab led by Dr. Omid Ardakanian.

Parts of Chapter 2 has been published as: A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, T. Dauphinee, "Migrating from Monolithic to Serverless: A FinTech Case Study", in *ICPE20: Companion of the ACM/SPEC International Conference on Performance Engineering (HotCloudPerf 2020)* [38].

Parts of Chapter 3 has been accepted as: A. Goli, N. Mahmoudi, H. Khazaei, O. Ardakanian, "A Holistic Machine Learning-Based Autoscaling Approach for Microservice Applications", in *11^{th} International Conference on Cloud Computing and Services Science* [39].

Parts of Chapter 4 has been published as: S. Gholami, A. Goli, C.P. Bezemer, H. Khazaei, "A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning", in *ICPE20: Proceedings of the ACM/SPEC International Conference on Performance Engineering* [35]. I was responsible for preparing the testbed, conducting the ZNN experiments, and developing the manuscript. S. Gholami was responsible for other parts of the work.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors, Hamzeh Khazaei and Omid Ardakanian, for their support and guidance throughout my graduate studies. It has been a privilege for me to work under the supervision of such knowledgeable, kind, and energetic professors.

I would like to thank all of my colleagues and friends from the PACS Lab and Sustainable Computing Labs for their help. I also would like to thank ATB Financial for providing funding and computational resources on the Google Cloud Platform for this research.

Last but by no means least, I would like to thank my parents for their unconditional love, support, and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Over the past decade, cloud computing has emerged as the mainstream model for delivering computational resources and online services. As can be seen in Figure 1.1, the cloud computing delivery model can be divided into three main categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). The control over the underlying infrastructure decreases as we go from IaaS to SaaS. IaaS refers to fundamental computing resources such as storage, networking, servers, and other raw resources. PaaS provides a pre-defined and cloud-based environment for developing and deploying custom applications. SaaS refers to all cloud-based applications which are available online and maintained by the service provider [10]. Public cloud providers such as Google, Amazon, Microsoft, and IBM offer these three types of cloud computing at a large scale.

Cloud computing provides a wide range of benefits. It reduces infrastructure costs, improves organizational agility, and makes capacity planning easier for organizations [31]. Companies and startups can avoid the huge capital expense for infrastructure ownership that they had to pay up-front in the past. With on-demand access to computational resources, they can rapidly work on new ideas, adapt themselves to market needs, and have a shorter time-to-market. Besides, the risk of overprovisioning and underprovisioning of resources will be mitigated [33].

Cloud computing has transformed the design, development, and deploy-

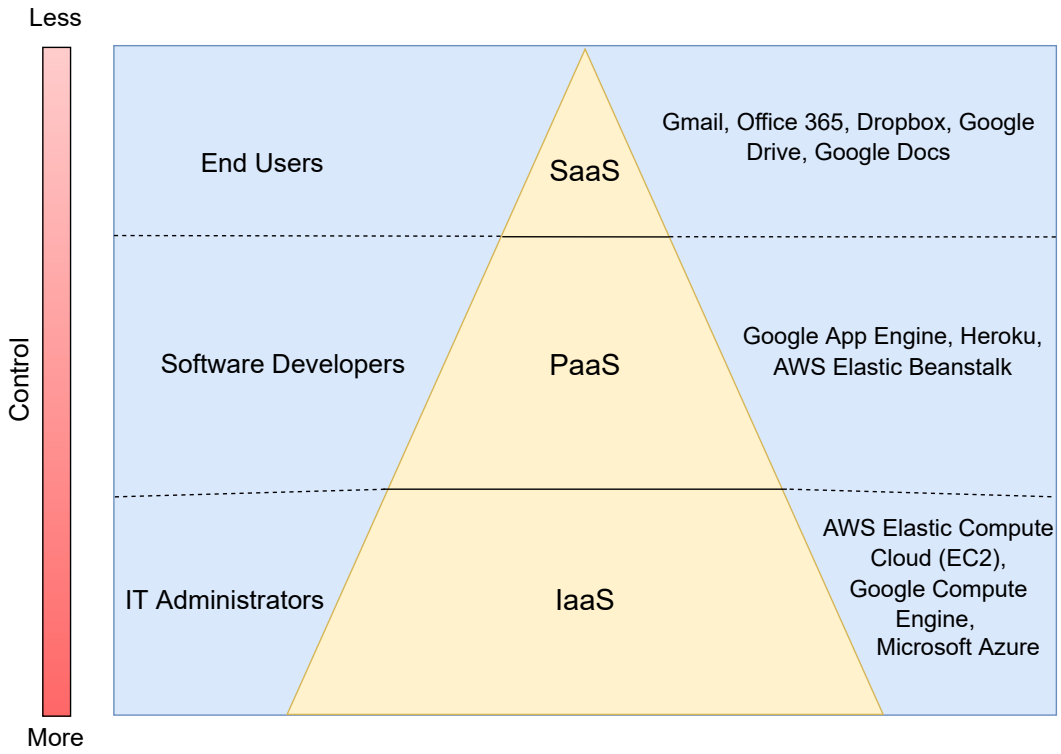


Figure 1.1: Cloud computing delivery model: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS)

ment of software systems. Nowadays, software engineers design software systems with scalability in mind thanks to cloud elasticity. They favour more scalable software architectures, such as service-oriented architectures [26]. In the development phase, developers save time by using cloud services for general tasks such as authentication, storage, and logging instead of reinventing the wheel every time they work on a new project. In the deployment phase, the elasticity of resources and pay-as-you-go cost model make the cloud an ideal place for deploying applications compared to on-premises servers. Moreover, infrastructure setup is not a manual process in the cloud and can be automated using infrastructure as code tools (IaC) (e.g., Chef, Puppet) [26]. In addition, implementing fault-tolerant mechanisms for server instances and data storage is no longer the developers' responsibility. Cloud providers take care of infrastructure and are responsible for maintaining and resolving hardware and software issues [72]. Consequently, developers are free from operational tasks and can focus solely on the product at hand.

1.2 Cloud Applications and Quality of Service

Applications that leverage the cloud computing delivery model are called *cloud applications* in general. As Figure 1.2 shows, based on the level of compatibility with the cloud environment, cloud applications can be further divided into three types: cloud-enabled, cloud-aware, and cloud-native. Cloud-enabled applications originally have been designed in a tightly coupled fashion for the on-premises static servers and later tweaked to be deployed on the cloud environment. Cloud-aware applications are in the next level after cloud-enabled applications. They interact with the cloud platform and take advantage of different cloud services such as load balancer, storage, service bus, etc. Cloud-native applications are in the next level and are fully designed and optimized for the cloud environment [70]. Cloud Native Computing Foundation (CNCF)[1] defines cloud-native as: "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil." Cloud-native applications are scalable, flexible, and platform agnostic. Microservice and serverless are two popular architectures which are in compliance with the cloud-native principles.

One fundamental concept regarding cloud services is quality of service (QoS). QoS is a measurement for the level of availability, reliability, and performance provided by the service residing in the cloud [13]. Availability is the probability that the service is up and provides the expected operation at a point in time. For example, six-nines or 99.9999% availability means that the service is available 99.9999% of the time and may be down only 0.0001% of the time. Reliability indicates the probability that the service provides the expected operation correctly and without failure over a stated time period [69].

[1] Cloud Native Foundation: `https://www.cncf.io`

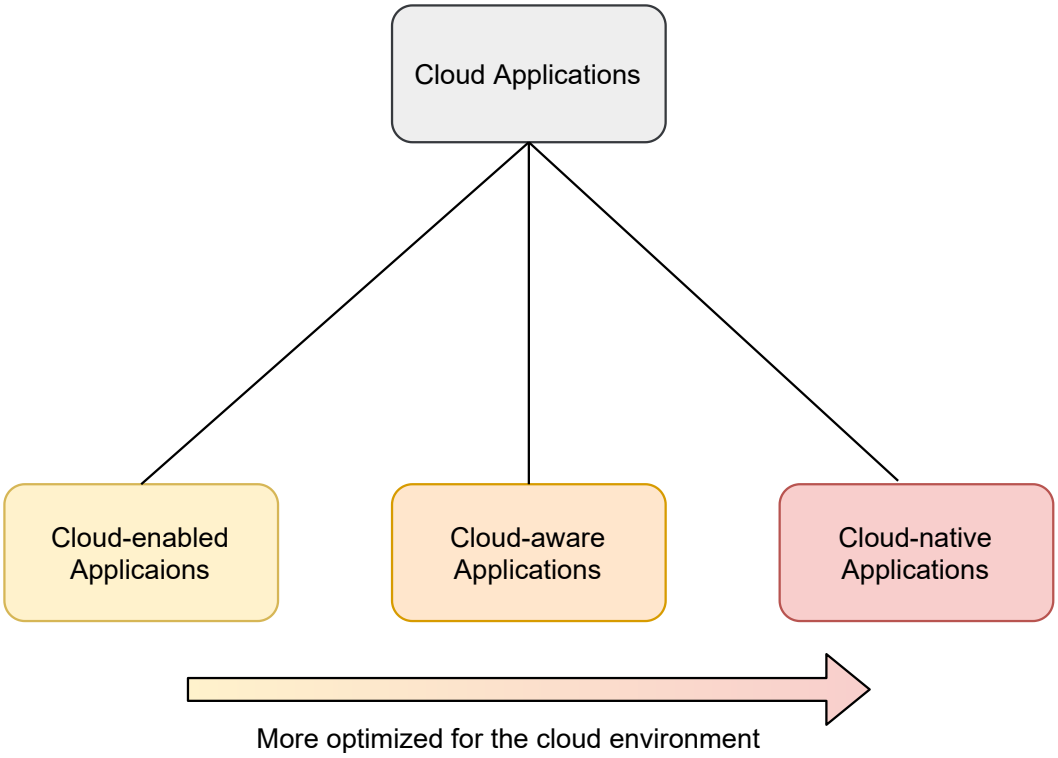


Figure 1.2: Cloud application types based on the level of compatibility with cloud environment.

Performance shows how well the service carries out the expected operation in terms of response time and throughput. Cloud providers should maintain the QoS at a certain level according to the service level agreement (SLA); otherwise, they will be penalized based on SLA terms. Moreover, failure to provide advertised QoS according to SLA can lead to customer dissatisfaction and financial loss of service consumers.

In this thesis, we aim to investigate different ways to improve and maintain performance as one of the QoS's dimensions. One of the most influential factors in having a performant application is compatibility with cloud environments. Applications designed based on cloud-native principles have a scalable architecture that can easily benefit from the elasticity of cloud resources to handle the high traffic load. Considering that the target application has a scalable architecture, the next step is to have an effective adaptation mechanism in place to maintain the application performance. With the increase in application load, the most straightforward way to maintain the application's performance

is increasing the application capacity by horizontal/vertical scaling. In horizontal scaling (aka scaling out), we increase the application capacity by adding more instances of the application/application components. In vertical scaling (aka scaling up), we increase the application capacity by adding more resources to the current application instances/application components. Cloud providers use autoscalers to automate the task of increasing and decreasing application capacity based on the application load. However, autoscaling may not be applicable in situations where our budget and computational resources are limited. In these situations, we can maintain the application performance by adapting our application and managing the available resources more efficiently.

Therefore, to improve the performance of cloud applications, we identify the following objectives:

- Redesigning applications with old architectures that suffer from poor scalability and migrate them to more scalable architectures that follow cloud-native principles. This objective is accomplished in Chapter 2.
- Developing an effective autoscaling mechanism for applications with scalable architectures. This objective is accomplished in Chapter 3.
- Developing a cost-effective approach for maintaining the performance in situations where autoscaling is not possible. This objective is accomplished in Chapter 4.

1.3 Summary of Contributions

In this thesis, we focus on performance as one of the QoS's dimensions and explore three ways for improving the performance of cloud applications.

This thesis makes the following contributions:

- We investigate how serverless computing can improve the performance of cloud applications by migrating a document processing system with monolithic architecture to a serverless architecture and comparing the performance of the serverless version with the monolithic version.

- We introduce a holistic autoscaling approach for cloud applications with microservice architecture that uses supervised learning to model the performance of each service in the application and the performance impact of services on each other.

- We present a cost-effective way for improving the performance of cloud applications using software multi-versioning. We show that containerization enables us to apply multi-versioning to specific components of a cloud application instead of the whole application to improve the performance.

1.4 Outline of the Thesis

The rest of this thesis is organized as follows. Chapter 2 discusses the migration of a FinTech cloud application with monolithic architecture to a serverless architecture. Chapter 3 presents the proposed machine learning-based approach for autoscaling of applications with microservice architecture. Chapter 4 presents our cost-effective approach for maintaining the performance of cloud applications at the level required by service level agreement using software multi-versioning. Chapter 5 concludes the thesis and presents some directions for future work.

Chapter 2

Migrating from Monolithic to Serverless: A FinTech Case Study

Serverless computing is steadily becoming the implementation paradigm of choice for a variety of applications, from data analytics to web applications, as it addresses the main problems with serverfull and monolithic architecture. In particular, it abstracts away resource provisioning and infrastructure management, enabling developers to focus on the logic of the program instead of worrying about resource management which will be handled by cloud providers. In this chapter, we consider a document processing system used in FinTech as a case study and describe the migration journey from a monolithic architecture to a serverless architecture. Our evaluation results show that the serverless implementation significantly improves performance while resulting in only a marginal increase in cost. The results presented in this chapter have been published in ICPE 2020 conference [38].

2.1 Introduction

Serverless computing is a new paradigm for developing applications and services, and a natural step in the evolution of cloud computing. It has emerged through the development of Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) technologies [51]. In this paradigm, the application logic is broken into functions that are stateless in nature and are left to the server-

less platform to manage. The granularity of serverless units has shifted from functions to stateless containers in recent years [7], [40]. Stateless containers are similar to stateless functions in the sense that they are triggered by an event, such as a HTTP request, and after the job execution completes, they disappear and do not save or carry any state information over to the next request. However, using stateless containers, developers are free to use the programming language and libraries of their choice to write code, which is not the case with stateless functions [41].

By supporting provisioning and autoscaling of resources while offering fine-grained pay-per-use billing [64], serverless computing has gained the attention of many enterprise and small-scale companies [5]. Recently, several machine learning and analytics applications have been successfully migrated to serverless computing. The Siren distributed machine learning framework [75] and Graphless serverless graph analysis toolkit [73] are examples of these applications.

Today, cloud providers, such as Amazon and Google, provide a comprehensive list of benefits and caveats for serverless application developers (see for example AWS Lambda [6] and Cloud Functions [27]). However, the serverless computing paradigm has a number of disadvantages which have been explored in the literature. One example is the potential performance degradation of applications with data intensive and communication dependent tasks and functions [46], [64], [73]. Thus, whether the transition to a serverless implementation makes sense, in terms of performance and cost, depends on the target application. This chapter aims to answer this question for a rudimentary financial technology (FinTech) application.

We present a CPU and data intensive FinTech application as a case study and discuss the migration of this application from its crude monolithic architecture to a distributed, performant serverless architecture. The actual names have been removed as requested by our collaborator. We evaluate the system with serverless architecture in terms of performance and cost, and compare it with the original system before migration. We present different challenges and pitfalls that we faced in our implementation journey and discuss how we

addressed them. For example, to overcome the performance issue of communication between services, we used non-blocking I/O, and to meet the resource constraints of the serverless platform, we divided machine learning models. In this work, instead of using stateless functions, we use stateless containers executed in a serverless manner by the Cloud Run platform [40]. The main contributions of this chapter are as follows:

- We migrate a real-world application with traditional monolithic architecture to serverless architecture.
- We address the gap in the literature on the challenges of migrating to serverless architecture by identifying the barriers and challenges in the process of migrating a monolithic document processing system to serverless architecture.
- We evaluate the new system with serverless architecture in terms of performance, scalability and cost, and compare it with the legacy system.

The rest of this chapter is organized as follows. Section 2.2 provides background information about the terminologies that are used throughout the chapter. Section 2.3 introduces the case study system which is used in this study. Section 2.4 explains the migration journey, motivation, challenges, and the proposed serverless architecture. Section 2.5 introduces the data set used in this work and evaluates the proposed serverless solution. Section 2.6 reviews related work on serverless computing, and Section 2.7 concludes the chapter.

2.2 Background

In this section, we provide background information and define the terminology used throughout this chapter.

2.2.1 Serverless Computing

In serverless computing, despite what its name suggests, we still have servers that work in the background, but they are not visible to cloud users [51]. The

cloud provider takes care of infrastructure management, allowing the developers to focus on the business logic of their applications [46]. Hence, as a result the developers do not need to worry about allocating resources and preventing under/over provisioning of resources. As the number of requests for a service increases, the serverless platform scales the number of running instances to accommodate the increased demand. Similarly, when there is no workload, it scales them down to zero. The cost model is also different from traditional computation resources in the cloud. In serverless platforms, users only pay for the time that they use a specific resource.

The serverless paradigm also brings several benefits to the cloud providers. Specifically, it benefits cloud providers by increasing the utilization of their servers that might not be appealing to users or platforms such as ARM and RISC-V to handle computations [51]. Moreover, multitenancy reduces the amount of idle infrastructure, thereby reducing the degree of under-utilization. These advantages make serverless appeal to cloud providers to the extent that today all main cloud providers offer serverless solutions, including Amazon AWS Lambda [6], Google Cloud Functions [27], and open source projects such as Apache OpenWhisk [12].

2.2.2 Monolithic Architecture

Monolithic architecture is a simple way of designing and implementing software systems. In monolithic architecture, all of the component are combined into a single tightly coupled piece of software. Applications with monolithic architecture have a single code base and are deployed as a single unit in the runtime environment. For brand new projects, this architecture may work well at the beginning, but as requirements evolve this architecture makes it more difficult for developers to adapt. This is one of the main disadvantages of monolithic architecture.

2.3 A FinTech Case Study

We focus on a document processing system which is commonly used by financial institutions and, in particular, by our FinTech collaborator as a case study. The document processing pipeline is a replacement for the laborious tasks of reading, classifying, and extracting information from financial documents such as bank statements, balance sheets, letters, etc. Historically, these tasks were done manually in banks and financial institutions but have been automated in the past couple of years.

The first version of this system is implemented in the Python programming language and has a monolithic architecture. As shown in Figure 2.1, this system has five main components:

- PDF to Image Converter: Converts each page of a PDF document to image so that it can be processed by other components. This component is implemented using the PDF2Image library [16].
- Page Classifier: Gets an image as input and classifies it into one of the predefined document classes. This component is implemented using the Keras library [24] with Tensorflow backend [62].
- Preprocessing: Applies image processing and enhancement techniques to the input image, making it ready for the OCR component. This component is implemented using the OpenCV library [18].
- Optical Character Recognition (OCR): Extracts the text content from the input image. This component is implemented using the Tesseract OCR Engine [68].
- Database: Indexes and stores the output of the document processing pipeline for each input document.

The output of this system is fed to a Key-Value extractor system which is capable of extracting key-value pairs that the financial institutions are interested in.

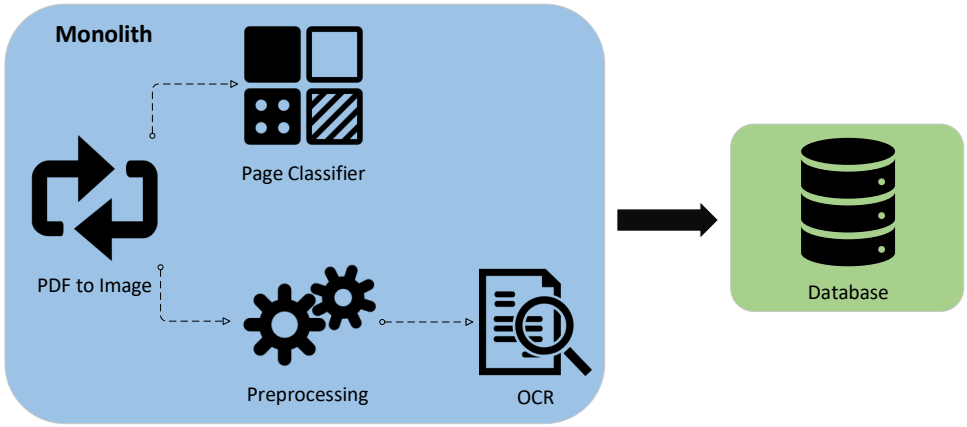


Figure 2.1: Architecture of the Original document processing system before migration. Arrows denote dependencies between modules.

2.4 Migration Journey

In this section, we walk through the migration journey and describe the challenges faced in each step. We start with the motivation behind the transition from a monolithic architecture to a serverless architecture. At last, we present the new serverless architecture.

2.4.1 Motivation

The original document processing system works well for a small number of input documents; however, it cannot handle a large number of documents in a reasonable amount of time within the non-functional requirements of large financial institutions. The two main issues with the original system are therefore speed and scalability.

We can imagine different ways to improve the performance of this system and make it process documents faster. For example, we can use multi-threading and parallel processing [78], but multi-threading may not be the most cost-effective solution because of the extra effort needed to tailor an existing software solution to a multi/many core execution environment. Furthermore, to achieve the best multi/many core performance, there is a need for certain infrastructure and hardware scale-ups which may not be afford-

able. For these reasons, the multi-threaded solutions could be less appealing compared to the serverless alternative.

For scalability, due to monolithic nature of the system, the only option for scaling without changing the implementation is vertical scaling by adding more resources to the machine that runs the application. This approach provides limited scalability and at some point cannot catch up with the growing demand [1]. Thus, the main motivation for migrating to the serverless architecture is to improve the performance and scalability of the system.

2.4.2 Breaking the Monolith

We started the migration journey by decoupling the system and breaking it down to a number of well-defined and loosely coupled services. In a loosely coupled system, components are separated so that they have the minimum knowledge of each other and changes in the implementation of one component do not impact other components. As depicted in Figure 2.1, it is possible to identify four well-defined services in the original monolithic architecture. These four services are: PDF to image conversion, page classification, preprocessing, and OCR.

After further consideration and taking into account the communication overhead and also the relationship between services, we decided to merge the preprocessing and OCR services into one service. This resulted in three primary services that together comprise the document processing pipeline.

Afterward, we started to package each service along with dependencies into a Docker container that follows the stateless principle of serverless computing. Each service resides in a container and listens for HTTP requests and input arguments. We trigger the stateless services via HTTP request and pass them the necessary arguments. The page classification service takes the image of a document as input and returns the prediction result as output. The input for the OCR service is also the image of the document; it returns the extracted text from the image.

However, after deploying this first architecture, we faced several challenges that led to modifying this architecture. In the next section, we review these

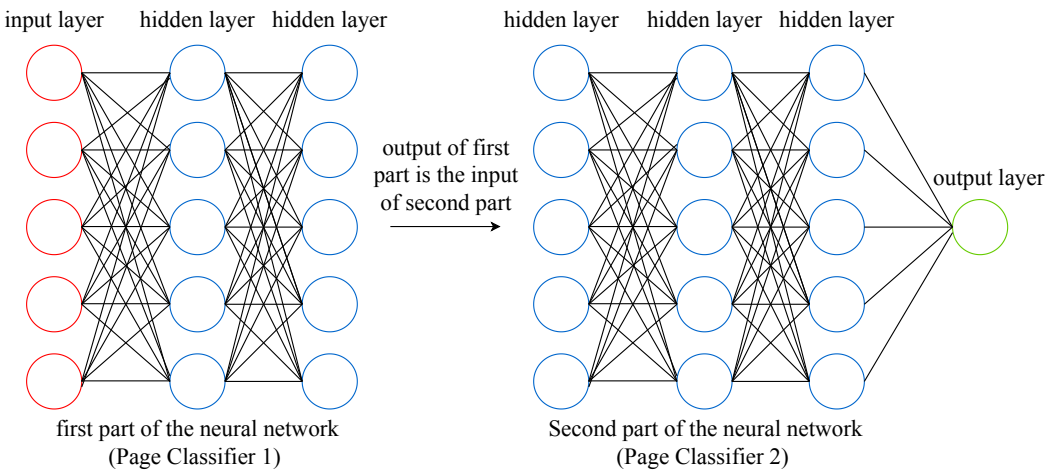


Figure 2.2: Structure of page classifier neural network after splitting.

challenges and changes that we implemented.

2.4.3 Migration Challenges

After implementing the first version of the serverless document processing pipeline based on the architecture in the previous section, we faced two main challenges. They can be attributed to the constraints that exist in the implementation and serverless solution provided by the cloud provider, which in our case was Google Cloud Platform (GCP).

The first challenge is the constrained resource of the serverless platform. This challenge is mainly due to the limitations and constraints that GCP imposes on instances of Cloud Run service. In particular, the maximum amount of memory that can be allocated to a Cloud Run service is 2 GB, but the page classification service needs more memory. To mitigate this challenge, we split the classifier component into two parts by breaking the trained neural network model, as shown in Figure 2.2. The first part performs a part of the prediction task and sends the output of the last hidden layer to the second part, which subsequently provides us with the final prediction, i.e., the document class. Therefore, the page classification service is replaced by two separate services Page Classifier 1 and Page Classifier 2.

The second challenge was related to the way that serverless services communicate with each other. In the first implementation, synchronous HTTP

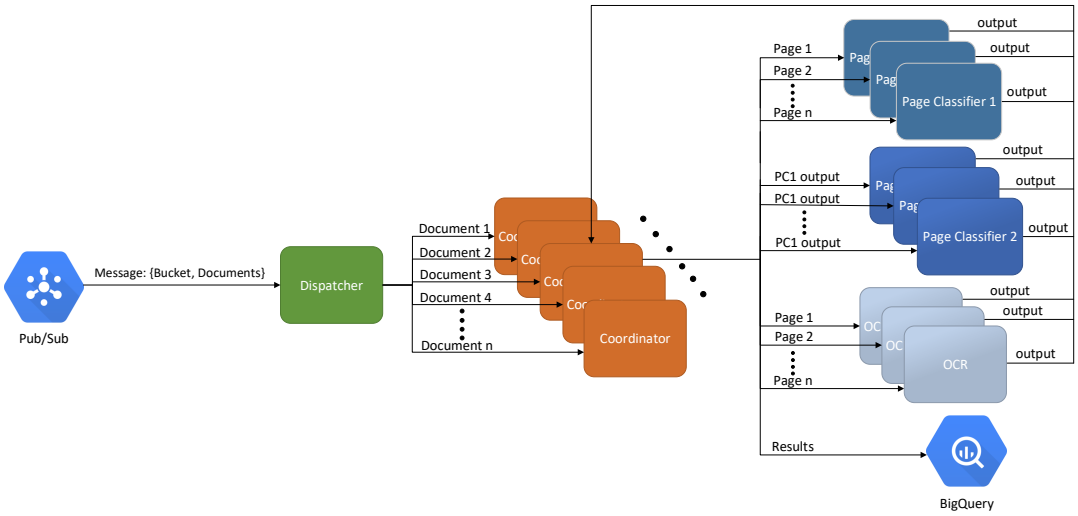


Figure 2.3: Serverless Architecture of Document processing pipeline.

requests were used for communication between services. So when we send an HTTP request in a synchronous manner, it blocks the execution of the program and waits for the response to arrive. In scenarios where we can execute other tasks at the same time, this paradigm diminishes the performance of the system and leads to wasting a considerable amount of computation time without doing useful computation, considering the cost model of serverless platform. To deal with this problem, we replaced synchronous HTTP requests with asynchronous HTTP requests for communication between services. Asynchronous HTTP requests follow the non-blocking programming paradigm in which after sending a request, the program can continue running other statements and do some useful task until the response arrives. Hence, in the document processing pipeline, when we send an asynchronous HTTP request, the program can continue sending other I/O requests, aggregate, and store the results of requests and complete the job in a shorter time in comparison with synchronous mode. We observe in the evaluation section that using asynchronous HTTP requests instead of synchronous HTTP requests improves the performance of the system tremendously. In the next section, we present the final serverless architecture after migration for the document processing pipeline.

2.4.4 Serverless Architecture

Figure 2.3 provides a high-level overview of the serverless architecture of the document processing pipeline after migration. In the new architecture, we have the following serverless services:

- Dispatcher: Receives a Pub/Sub message that contains the bucket name and the list of document names. It then assigns each document to an instance of the coordinator service.
- Coordinator: Plays the role of an orchestrator in the pipeline. This service is responsible for executing the pipeline logic on a single document, aggregating the results, and storing the results in BigQuery.
- Page Classifier 1: Gets an image as input and performs the first part of the classification task and sends the last hidden layer output to the coordinator service.
- Page Classifier 2: The input for this service is the output of Page Classifier 1. It executes the second part of the classification task and returns the final prediction to the coordinator service.
- OCR: Gets an image as input, extracts the text content from the image after preprocessing, and sends it to the coordinator service.

In this architecture, we leverage the inherent parallelism in the serverless platform which is the ability to launch a new service instance per request to speedup the document processing pipeline. As Figure 2.3 shows, in this new architecture, the coordinator service gets a document as input and splits the document into pages. Next it feeds each page to the Page Classifier 1 and OCR services at the same time using asynchronous HTTP requests. This means that each page is sent to a separate instance of OCR and Page Classifier 1 services and processed in parallel with other pages; this leads to higher throughput and increased performance.

Page Classifier 1 carries out the first part of the whole classification job and sends the output of the last hidden layer to the Page Classifier 2 through

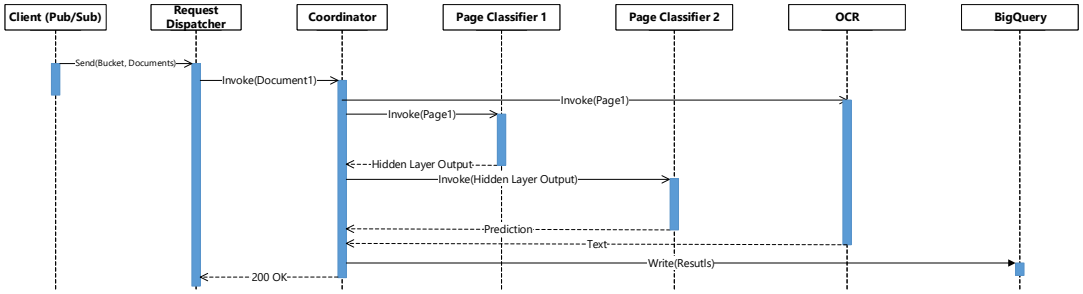


Figure 2.4: Sequence diagram of serverless document processing pipeline for one document with one page.

the coordinator service. Page Classifier 2 carries out the rest of the classification job and returns the final prediction to the coordinator service. In the OCR service, each page is preprocessed first and the text content is extracted and returned to the coordinator service. The coordinator service receives the results from all services, aggregates and stores them in BigQuery.

To simply show the interaction between different serverless services in the new architecture, we pass a single document with one page to the pipeline as depicted in Figure 2.4.

2.5 Evaluation

In this section, we evaluate the serverless document processing pipeline in terms of performance, scalability, and cost, and compare it with the old monolithic version.

2.5.1 Dataset Description

Due to privacy issues regarding financial documents, we cannot publicly share the dataset of financial documents that we used in our system; instead we take advantage of the publicly available RVL-CDIP dataset for classifier training and performance/cost evaluation of the new system[1][45]. This dataset contains 400,000 document images in 16 different classes. These 16 classes are letter, form, email, handwritten, advertisement, scientific report, scientific

[1]https://www.cs.cmu.edu/ aharley/rvl-cdip

publication, specification, file folder, news article, budget, invoice, presentation, questionnaire, resume, and memo. We stitch random samples from RVL-CDIP together to form a synthetic dataset of PDFs between 7-15 pages long.

2.5.2 Experimental Setup

We deploy each service in the proposed serverless architecture on GCP as a Cloud Run service. 2 GB of memory is allocated to each instance of the service. We also run the monolithic version on two separate virtual machines with different specifications on GCP to observe the effect of adding more resources on the scalability and performance of the monolithic version. The first virtual machine has 4 vCPU and 15 GB of memory, while the second one has 96 vCPU and 360 GB of memory.

We conduct two experiments. In the first experiment, we compare the serverless architecture with the monolithic one in terms of performance and cost by benchmarking both systems on a set of 100 documents that have between 7 and 15 pages. In the second experiment, we evaluate the performance and scalability of the new architecture as the number of input documents increases. We repeat each experiment three times and ignore the result of the first run to eliminate the cold start effect. Hence, we report the average of the results of the second and third runs.

2.5.3 Results

Table 2.1 shows the performance and cost results obtained for the old system and the new system on 100 documents in the first experiment. It can be readily seen that the serverless version is about 93x faster than the monolithic version; it processes the documents in about four minutes, whereas it takes about six hours for the monolithic version with 4 vCPU and 15 GB of memory to finish the same job. Turning our attention to cost, the monolithic version with 4 vCPU and 15 GB of memory turns out to be about $1.2 cheaper than serverless version. Note that we have not taken into account the idle time for the virtual machine in our cost calculations. The cost for the virtual machine

Table 2.1: Comparing the monolithic architecture with serverless architecture in terms of performance and cost

Architecture	**Spec**	**Turnaround Time**	**Cost**
Monolithic (Single VM)	4 vCPU, 15 GB (n1-standard-4)	$\sim$ 6.27 h	$\sim$ \$1.23
Monolithic (Single VM)	96 vCPU, 360 GB (n1-standard-96)	$\sim$ 5.07 h	$\sim$ \$23.12
Serverless (Cloud Run)	1 vCPU, 2 GB	$\sim$ 4.05 min	$\sim$ \$2.43

is only for the duration that it processes the document, so if we consider the idle time in the cost calculation (which is the case in the real-world setting), it would become more expensive than the serverless solution.

We also repeated the first experiment for the monolithic version on a more powerful virtual machine with 96 vCPU and 360 GB of memory to understand to what extent we can improve the performance and reduce the turnaround time by adding more resources. As expected, adding more resources did not greatly improve the performance despite increasing the cost dramatically. We attribute this to the sequential execution of the monolithic version.

In the second experiment, we change the number of input document to the serverless pipeline from 1 to 500 documents to measure the performance of the new system on different loads. Figure 2.5 illustrates the turnaround time as we increase the number of input documents. The x-axis of this figure shows the total number of documents submitted to the system where each document consists of multiple pages (11 pages on average).

Due to the limit set by GCP, each Cloud Run service can currently scale to a maximum of 1000 instances simultaneously. Thus, the number of instances varies between 0 to 1000 based on the input workload. As we invoke some services such as Page Classifier 1 for each page, at some point, these services reach this instance limit, causing the next requests to experience a waiting time before getting service. Therefore, we observe an almost linear increase in the turnaround time as the number of documents increases.

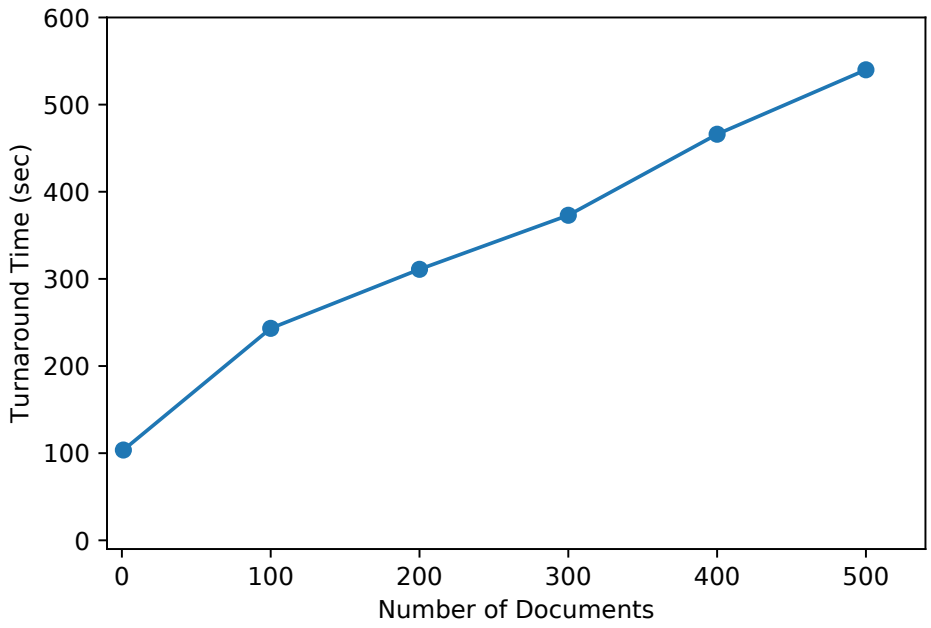


Figure 2.5: The turnaround time of jobs in the serverless document processing pipeline.

2.6 Related Work

Maintaining the application performance under heavy loads demands a scalable architecture. For applications with traditional software architectures, this goal can be achieved by migrating to more scalable architectures such as serverless architecture. In recent years, due to the appealing characteristics of FaaS serverless solutions such as implicit autoscaling and fine grained billing benefits, many companies and developers have migrated their solutions to the serverless architecture to benefit from these advantages and provide better services to their costumers. Such companies include Reuters, iRobot, Autodesk and many more that reportedly use AWS Lambda to better serve their customers and internal processes [8].

Toader et al. [73] proposed a detailed implementation of a serverless graph processing framework implemented with AWS Lambda which benefits from automated resource autoscaling and provisioning. Graphless abstracts away resource management and configuration from the users. This will benefit end

users who are not familiar with HPC concepts. However, the authors have shown that because of the variability of network characteristics under certain communication intensive workloads, the Graphless efficiency degrades. Wang et al. [75] proposed a serverless framework for machine learning tasks, called Siren, which is fully asynchronous and achieves different levels of parallelism and elasticity. Siren, through stateless serverless functions, much like Graphless eliminates the burden of resource management and scaling machine learning algorithms imposed on the end users by the current well established serverfull frameworks. The Siren framework is deployed on the AWS Lambda serverless platform. Kurz et al.[57] leveraged the inherent parallelism in the serverless execution model and introduced a serverless architecture to estimate double machine learning models. Adzic et al. [5] discussed economic and architectural benefits of serverless computing and study two real world services, namely MindMup and Yubl [5], that have been migrated to and adopted serverless computing. They discussed how MindMup and Yubl could benefit from serverless deployment; these benefits include reduced time to feature delivery and time to market for developers and faster request processing for customers. However, it is mentioned that serverless platforms are not well suited for mission critical and time sensitive tasks [5].

2.7 Conclusion

With the advent of serverless computing, several monolith applications which were previously developed for a single-core or multi-core execution environment are implemented from scratch following the serverless paradigm to take advantages of auto-scaling and automated resource provisioning. In this chapter, by observing the gap in the literature and the lack of studies concerning the challenges of migration to serverless and performance of the serverless implementation of financial services, we present the migration journey of a FinTech application from its monolithic form to a high-performance serverless implementation. Based on our evaluations, the proposed serverless implementation outperforms the previous monolith serverfull implementation by 93 times with

a marginal increase in cost. Our cost calculation does not take into account the under utilization of the serverfull infrastructure and the investment required to build the serverless system. In conclusion, the serverless implementation provides unparalleled speedup and performance improvement over the serverfull implementation without making drastic changes to the software design.

Chapter 3

A Holistic Machine Learning-Based Autoscaling Approach for Microservice Applications

Microservice architecture is the mainstream pattern for developing large-scale cloud applications as it allows for scaling application components on demand and independently. By designing and utilizing autoscalers for microservice applications, it is possible to improve their availability and reduce the cost when the traffic load is low. In this chapter, we propose a novel predictive autoscaling approach for microservice applications which leverages machine learning models to predict the number of required replicas for each microservice and the effect of scaling a microservice on other microservices under a given workload. Our experimental results show that the proposed approach in this work offers better performance in terms of response time and throughput than HPA, the state-of-the-art autoscaler in the industry, and it takes fewer actions to maintain a desirable performance and quality of service level for the target application. The results presented in this chapter will be published in CLOSER 2021 conference [39].

3.1 Introduction

Microservice is the most promising architecture for developing modern large-scale cloud software systems [29]. It has emerged through the common patterns adopted by big tech companies to address similar problems, such as scalability and changeability, and to meet business objectives such as reducing time to market and introducing new features and products at a faster pace [65]. Traditional software architectures, such as monolithic architecture, are not capable of providing such level of efficiency [29]. Companies like SoundCloud, LinkedIn, Netflix, and Spotify have adopted the microservice architecture in their organization in recent years and reported success stories of using it to meet their non-functional requirements [20], [48], [63], [65].

In the microservice paradigm, the application is divided into a set of small and loosely-coupled services that communicate with each other through a message-based protocol. Microservices are autonomous components which can be deployed and scaled independently.

The Microservice architecture provides a wide range of benefits. In terms of software development, microservice architecture provides higher agility, better comprehensibility, and simplified testability. New features can be added to the system more frequently without changing the whole system or too much configuration. In the development process, each team usually works on a service independent of other teams, which leads to an increase in the speed of development. The language and tools can also be different in different teams so that the right tool and programming language can be used according to the service requirement. The whole system is also easier to maintain thanks to the replaceability of components. Microservice architecture also improves the team's understanding of the whole system. In the case of monolithic architecture, it is not easy to understand the whole system, and team members usually know just about the components that they are involved in their development process. From the performance perspective, microservice systems have better availability and runtime scalability compared to monolithic systems since each service can be managed and scaled independently [65], [71].

One of the key features of the microservice architecture is *autoscaling*. It enables the application to handle an unexpected demand growth and continue working under pressure by increasing the system capacity. While different approaches have been proposed in the literature for autoscaling of cloud applications [32], [56], [58], [60], [67], most related work is not tailored for the microservice architecture [67]. This is because a holistic view of the microservice application is not incorporated in most related work; hence each service in the application is scaled separately without considering the impact this scaling could have on other services. To remedy the shortcoming of existing solutions, a more effective and intelligent autoscaler can be designed for microservice applications, a direction we pursue in this chapter.

We introduce Waterfall autoscaling (hereafter referred to as Waterfall for short), a novel approach to autoscaling microservice applications. Waterfall takes advantage of machine learning techniques to model the behaviour of each microservice under different load intensities and the effect of services on one another. Specifically, it predicts the number of required replicas for each service to handle a given load and the potential impact of scaling a service on other services. This way, Waterfall avoids shifting load or possible bottlenecks to other services and takes fewer actions to maintain the application performance and quality of service metrics at a satisfactory level. The main contributions of our work are as follows:

- We introduce data-driven performance models for describing the behaviour of microservices and their mutual impacts in microservice applications.
- Using these models, we design Waterfall which is a novel autoscaler for microservice applications.
- We evaluate the efficacy of the proposed autoscaling approach using Teastore, a reference microservice application, and compare it with a state-of-the-art autoscaler used in the industry.

The rest of this chapter is organized as follows. Section 3.3 reviews re-

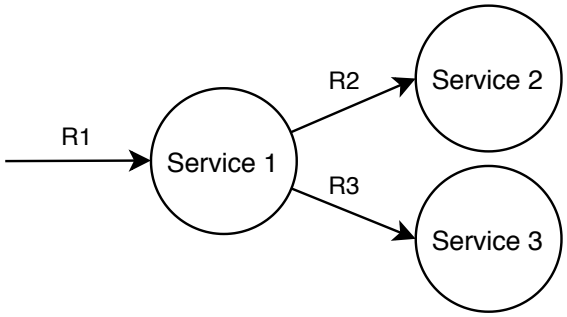


Figure 3.1: Interaction of services in an example microservice application.

lated work on autoscaling and Section 3.2 provides a motivating example. Section 3.4 presents the proposed machine learning-based performance models for microservice applications. Section 3.5 describes the design of Waterfall autoscaler. Section 3.6 evaluates the proposed autoscaling technique, and Section 3.7 concludes the chapter.

3.2 Motivating Scenario

A microservice application usually consists of multiple services interacting with each other to accomplish their job. The rate at which a service sends requests to the downstream services depends on the rate at which it receives requests and the amount of resources available for processing these requests. Thus, scaling a service that may invoke a group of other services might subsequently change the load on those services. Consider the interaction between three services in an example microservice application depicted in Figure 3.1. Service 1 calls Service 2 and Service 3 to complete some tasks. If Service 1 is under heavy load (R1), scaling Service 1 would cause an increase in the load observed by Service 2 (R2) and Service 3 (R3). If we predict how scaling Service 1 degrades the performance of Service 2 and Service 3, we can avoid the shift in the load and a possible bottleneck from Service 1 to Service 2 and Service 3 by scaling Service 2 and Service 3 proactively at the same time as Service 1.

To further examine the cascading effect of scaling a service in a microservice application on other services, we conducted an experiment using an ex-

ample microservice application called Teastore [55]. Teastore[1] is an emulated online store for tea and tea-related products. It is a reference microservice application developed by the performance engineering community to provide researchers with a standard microservice application that can be used for testing and evaluating research in different areas such as performance modelling, cloud resource management, and energy efficiency analysis [55]. Figure 3.2 shows services in the Teastore application and their interactions. The solid lines show the dependencies between services, and dashed lines indicate that the service call happens only once at startup time. Teastore includes five primary services: Webui, Auth, Persistence, Recommender, and Image. Webui is the front-end service that users interact with and is responsible for rendering the user interface. Auth stands for authentication; it verifies the user's credentials and session data. The Persistence service interacts with the database and performs create, read, update, and delete (CRUD) operations. The Recommender service predicts the user preference for different products and recommends appropriate products to users using a collaborative filtering algorithm. The Image service provides an image of products in different sizes. In addition to main services, Teastore has another component named Registry, which is responsible for service registration and discovery.

As can be seen in Figure 3.2, depending on the request type, the Webui service may invoke Image, Persistence, Auth, and Recommender services. We generate a workload comprising different types of requests so that Webui service calls all of these four services. Keeping the same workload intensity, we increased the number of replicas for the Webui service from 1 to 5 and monitored the request rate of Webui in addition to the downstream rate of the Webui service to other services that each has one replica. For the two services m and n, we define the request rate of service m, and the downstream rate of service m to service n as:

$$\text{Request Rate}(m) = \text{number of requests service } m \text{ receives per second} \quad (3.1)$$

[1] `https://github.com/DescartesResearch/TeaStore`

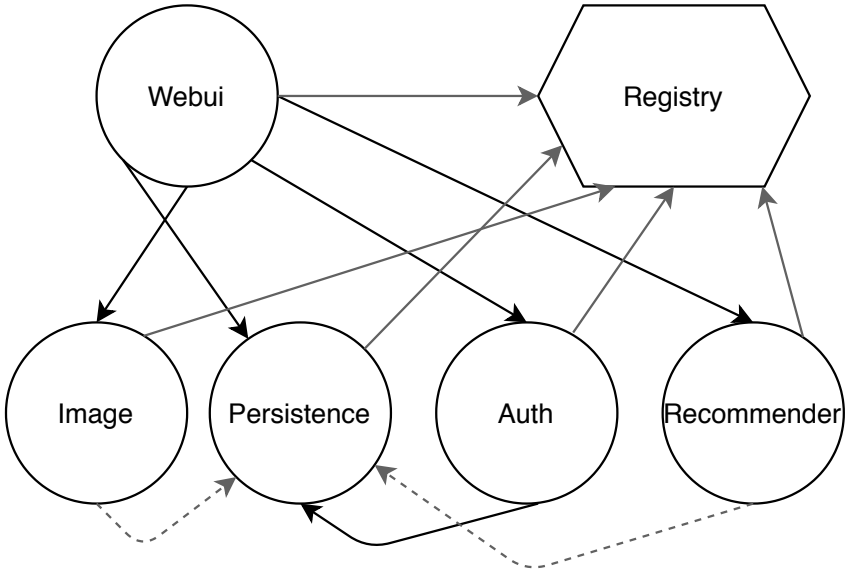


Figure 3.2: Architecture of the Teastore application.

$$\text{Downstream Rate}(m, n) = \text{number of requests service } m \text{ sends to service } n \text{ per second} \tag{3.2}$$

For instance, in Figure 3.1, *Request Rate(Service 1)* is equal to $R1$ and *Downstream Rate(Service 1, Service 2)* is equal to $R2$.

Table 3.1 shows the request rate of Webui and its downstream rate to each service for the different number of replicas. As can be seen, scaling the Webui service leads to an increase in its request rate, which in turn increases the downstream rate of the Webui service to other services. Therefore, under heavy load, scaling the Webui service increases the load on the other four services. Figure 3.3 shows the results of our experiment. The left plot and right plot show the request rate and total downstream rate of the Webui service for different number of replicas, respectively. Error bars indicate the 95% confidence interval.

The cascading effect of microservices on each other motivates the idea of having an autoscaler that takes this effect into account and takes action accordingly. Autoscalers that consider and scale different services in an application independently are unaware of this relationship, thereby making premature decisions that could lead to extra scaling actions and degradation in the quality

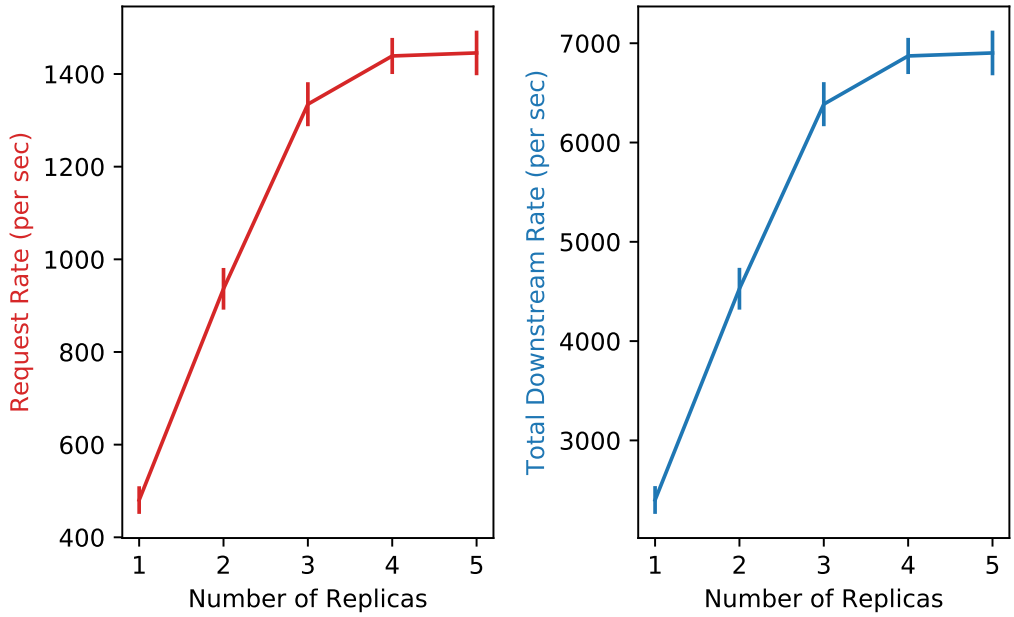


Figure 3.3: Request rate and total downstream rate of Webui under the same load intensity for different numbers of replicas.

of service of the application. In this work, we introduce a novel autoscaler to address this shortcoming in existing autoscalers.

3.3 Related Work

Autoscaling is a widely used and well-known concept in cloud computing, mainly due to the elasticity and pay-as-you-go cost model of cloud services. With the shift in the runtime environment of microservice applications from

Table 3.1: Request Rate and Downstream Rate of the Webui service to each service. The number of replicas for the Webui service changes from 1 to 5.

Monitored Metric	**1**	**2**	**3**	**4**	**5**
Request Rate(Webui)	480	936	1334	1438	1445
Downstream Rate(Webui,Persistence)	1083	2108	3005	3239	3253
Downstream Rate(Webui,Auth)	482	938	1337	1441	1447
Downstream Rate(Webui,Image)	562	1094	1559	1680	1688
Downstream Rate(Webui,Recommender)	121	235	334	360	362

bare-metal servers to more fine-grained environments, such as virtual machines and containers in the cloud, autoscaling has become an indispensable part of microservice applications. The autoscalers can be categorized based on different aspects from the underlying technique to the decision making paradigm (e.g., proactive or reactive) and the scaling method (e.g., horizontal, vertical, or hybrid) [67]. Based on the underlying technique, autoscalers can be classified into five categories: rule-based methods, application profiling methods, analytical modelling methods, and machine learning-based methods.

Rule-based autoscalers act based on a set of predefined rules to scale and estimate the amount of necessary resources for provisioning. This type of autoscalers is common in the industry and usually serves as the baseline [67]. Products such as Amazon AWS Autoscaling service [11] and Kubernetes Horizontal Pod Autoscaler (HPA) [56] fall into this group. Wong et al. [58] proposed two rule-based autoscalers similar to Kubernetes HPA for microservices, namely $\text{HyScale}_{\text{CPU}}$ and $\text{HyScale}_{\text{CPU+Mem}}$. $\text{HyScale}_{\text{CPU}}$ uses both horizontal and vertical scaling to scale each microservice in the target application separately based on CPU utilization. It gives priority to vertical scaling and applies horizontal scaling only if the required amount of resources cannot be acquired using vertical scaling. $\text{HyScale}_{\text{CPU+Mem}}$ operates similarly except that it considers memory utilization in addition to CPU utilization for making the scaling decision. Although rule-based autoscalers are easy to implement, they typically need expert knowledge about the underlying application for tuning the thresholds and defining the scaling policies [67].

Application profiling methods measure the application capacity with a variety of configurations and workloads and use this knowledge to determine the suitable scaling plan for a given workload and configuration. For instance, Fernandez et al. [32] proposed a cost-effective autoscaling approach for single-tier web applications using heterogeneous Spot instances [9]. They used application profiling to measure the processing capacity of the target application on different types of Spot instances for generating economical scaling policies with a combination of on-demand and Spot instances.

In autoscalers with analytical modelling, a mathematical model of the sys-

tem is used for resource estimation. Queuing models are the most common analytical models used for performance modelling of applications in the cloud. In applications with more than one component, such as microservice applications, a network of queues is usually considered to model the system. Gias et al. [37] proposed a hybrid (horizontal+vertical) autoscaler for microservice applications based on a layered queueing network model (LQN) named ATOM. ATOM uses a genetic algorithm in a time-bounded search to find the optimal scaling strategy. The downside of modelling microservice applications with queuing network models is that finding the optimal solution for scaling is computationally expensive. Moreover, in queueing models, measuring the parameters such as service time and request mix is non-trivial and demands a complex monitoring system [67].

Search-based optimization methods use a meta-heuristic algorithm to search the state space of system configuration for finding the optimal scaling decision. Chen et al. [22] leveraged a multi-objective ant colony optimization algorithm to optimize the scaling decision for a single-tier cloud application with respect to multiple objectives.

Machine learning-based autoscalers leverage machine learning models to predict the application performance and estimate the required resources for different workloads. Machine learning techniques can be divided into regression and reinforcement learning methods. Regression-based methods usually find the relationship between a set of input variables and an output variable such as resource demand or a performance metric. Wajahat et al. [74] proposed a regression-based autoscaler for autoscaling of single-tier applications. They considered a set of monitored metrics to predict the response time of the application, and based on predictions, they increased or decreased the number of virtual machines assigned to the application on OpenStack. Jindal et al. [50] used a regression model to estimate the microservice capacity (MSC) for each service in a microservice application. MSC is the maximum number of requests that a microservice with a certain number of replicas can serve per second without violating the service level objective (SLO). They obtained this value by sandboxing and stress-testing each service for several configuration

deployments and then fitting a regression model to the collected data. In reinforcement learning approaches, an agent tries to find the optimal scaling policy for each state of the system (without assuming prior knowledge) through interaction with the system. Iqbal et al. [49] leveraged reinforcement learning to learn autoscaling policies for a multi-tier web application under different workloads. They identified the workload pattern from access logs and learned the appropriate resource allocation policy for a specific workload pattern so that SLO is satisfied and resource utilization is minimized. The drawback of reinforcement learning methods is the poor performance of autoscalers at the early stages of deployment because it takes some time for the reinforcement learning model to learn the optimal policy. Moreover, machine learning has been used for workload prediction in proactive autoscaling. These methods use time series forecasting models to predict the future workload and provision the resources ahead of time based on the prediction for the future workload. Coulson et al. [28] used a stacked LSTM [47] model to predict the composition of the next requests and scale each service in the application accordingly. Abdullah et al [2] introduced a proactive autoscaling method for microservices in fog computing micro data centers. They predict the incoming workload with a regression model using different window sizes and identify the number of containers required for each microservice separately. The main problem with these methods is that they can lead to dramatic overprovisioning or underprovisioning of resources [67] owing to the uncertainty of workload arrivals, especially in the news feed and social network applications.

3.4 Predicting Performance

This section presents machine learning models adopted for performance modelling of microservice applications. These models are at the core of our autoscaler for predicting the performance of each service and possible variations in performance as a result of scaling another service. Hence, we utilize two machine learning models for each microservice which are described in the following sections.

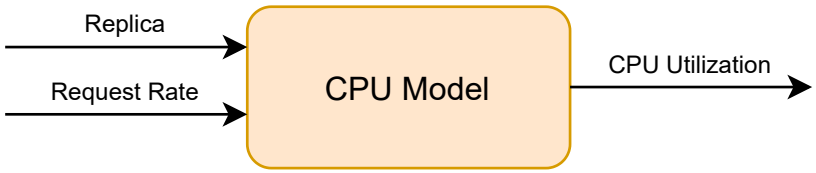


Figure 3.4: Input features and the predicted value of the CPU model.

3.4.1 Predictive Model for CPU Utilization

The *CPU Models* captures the performance of microservices in an application in terms of CPU utilization. CPU utilization is a good proxy for estimating the workload of a microservice [43]. Therefore, we use the average CPU utilization of the microservice replicas as the performance metric for scaling decisions. Depending on the target performance objective, this metric can be replaced with other metrics, such as response time and message queue metrics.

As Figure 3.4 demonstrates, the CPU Model takes the number of service replicas and the request rate of service as input features and predicts the service's average CPU utilization. In other words, this model can tell us what would be the average CPU utilization of service under a specific load.

3.4.2 Predictive Model for Request Rate

The *Request Model* predicts the new request rate of a microservice after scaling and changing the number of service replicas. As shown in Figure 3.5, we feed the current number of service replicas, the current average CPU utilization, the current request rate, and the new number of service replicas as input features to the Request Model to predict the new request rate for the service. The current replica, current CPU utilization, and current request rate describe the state of the service before scaling. The new replica and new request rate reflect the state of the service after scaling. We use the output of the Request Model for a given service to calculate the new downstream rate of that service to other services. Thus, the Request Model helps us predict the effect of scaling a service on other services.

As we discussed in Section 3.2, any changes in the request rate of a service in a microservice application might lead to changes in the downstream rate

Figure 3.5: Input features and the predicted value of the request model.

of that service to other services. However, we observed that under the same workload intensity, when we scale a service, the downstream rate of that service to another service changes linearly with respect to its request rate. For instance, we used the results from Section 3.2 in Table 3.1 and divided the downstream rate of Webui service to other services by its request rate and produced Table 3.2. Consequently, when we scale a service, if we have the new request rate after scaling, we can calculate its new downstream rate to other services. We achieve this goal through Request Model. For example, according to Table 3.1 for one replica *Request Rate(Webui)* $\approx$ *480* and *Downstream Rate(Webui,Persistence)* $\approx$ *1083*. Moreover, from Table 3.2 we know that for all replica counts, *Downstream Rate(Webui,Persistence) / Request Rate(Webui)* $\approx$ *2.25*. Therefore, if we scale out the Webui service to two replicas and have the new value for *Request Rate(Webui)* as *936*, we can estimate the new *Downstream Rate(Webui,Persistence)* by multiplying the new *Request Rate(Webui)* by *2.25* which will be *936 * 2.25* $\approx$ *2106*. The reason for the difference between the calculated value (2106) and the real value (2108) for the new *Downstream Rate(Webui,Persistence)* is that numbers in Table 3.1 and Table 3.2 are rounded due to lack of space.

3.4.3 Data Collection

To train CPU Model and Request Model for each microservice, we needed to collect two datasets per microservice. The data collection for each microservice is performed independent of other services. We deploy enough number of replicas from other services to avoid any limitations imposed by other services on the target service for data collection.

The dataset for CPU Model includes three metrics: the number of replicas,

Table 3.2: The ratio of Downstream Rate values of Webui service to its Request Rate for different number of replicas under the same workload intensity.

Downstream Rate/Request Rate	**1**	**2**	**3**	**4**	**5**
Downstream Rate(Webui,Persistence)/ Request Rate(Webui)	2.25	2.25	2.25	2.25	2.25
Downstream Rate(Webui,Auth)/ Request Rate(Webui)	1.00	1.00	1.00	1.00	1.00
Downstream Rate(Webui,Image)/ Request Rate(Webui)	1.17	1.17	1.17	1.17	1.17
Downstream Rate(Webui,Recommender)/ Request Rate(Webui)	0.25	0.25	0.25	0.25	0.25

the request rate per second, and the average CPU utilization of replicas. Each data point results from applying a workload with a fixed number of threads for 12 minutes to the front-end service. At the end of each run, we collect each metric's values during this period and use their mean as the value of the metric for that data point. Note that we ignored data values for the first and last minute of each run to exclude the warm-up and cool-down periods. We consider a different number of replicas for the target service, and for each number of replicas, we change the number of threads to increase the number of requests until we reach the saturation point for that specific number of replicas. For instance, for one replica of an example service, we apply the workload with 1, 2, 3, 4, and 5 threads, resulting in five data points.

The Request Model dataset contains five metrics: the current number of replicas, the current request rate per second, the current average CPU utilization of replicas, the new number of replicas, and the new request rate per second. Each data point for this dataset results from the merging of two runs from the CPU Model dataset with the same number of threads but a different number of replicas. For example, we merge the result for the run with one replica and five threads with the result for two replicas and five threads to generate a data point for the Request Model dataset. More specifically, we get the current replica, current CPU utilization, and current request rate from the

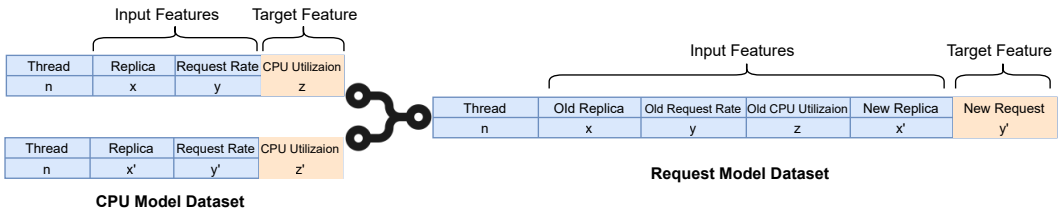


Figure 3.6: The construction of datasets for CPU Model and Request Model. Request Model dataset is built by merging data points from the CPU Model dataset.

first run and the new replica and new request rate from the second run.

Figure 3.6 shows an example data point for CPU Model and Request Model datasets. The data point for Request Model is a combination of two runs that have n threads with x and x' replicas, respectively.

3.4.4 Model Training Results

We trained CPU Model and Request Model for all microservices in the Teastore application using datasets created from collected data. Each dataset was split into training and validation sets. The training sets and validation sets contain 80% and 20% of data, respectively. We used Linear Regression, Random Forest, and Support Vector Regressor algorithms for the training process and compared them in terms of mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), and R^2 score. Table 3.3 and Table 3.4 show the results for CPU Model and Request Model of each microservice, respectively. As can be seen from the results, Support Vector Regressor and Random Forest provide lower MAE, MSE, RMSE, and higher R^2 score for CPU Model and Request Model compared to Linear Regression. Currently, we use offline learning to train machine learning models, but our approach can be adapted to leverage online learning as well.

3.5 Waterfall Autoscaler

In this section, we present the autoscaler we designed using the performance models described in Section 3.4. We first outline the architecture of Waterfall

Table 3.3: The accuracy and R^2 score of CPU Model for different services using Linear Regression (LR), Random Forest (RF), and Support Vector Regressor (SVR).

Service	Linear Regression				Random Forest				SVR			
	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score
Webui	4.97	45.32	6.73	92.21	3.67	18.57	4.31	96.81	1.43	3.07	1.75	99.47
Persistence	4.12	27.55	5.25	94.03	3.26	17.02	4.13	96.31	0.88	1.91	1.38	99.59
Auth	4.40	37.39	6.11	94.82	4.26	34.45	5.87	95.23	1.73	6.45	2.54	99.11
Recommender	2.62	12.42	3.52	92.94	1.39	4.23	2.06	97.60	1.38	5.00	2.23	97.16
Image	3.81	20.12	4.49	96.87	3.61	21.09	4.59	96.72	1.54	3.45	1.86	99.50

Table 3.4: The accuracy and R^2 score of Request Model for different services using Linear Regression (LR), Random Forest (RF), and Support Vector Regressor (SVR).

Service	Linear Regression				Random Forest				SVR			
	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score
Webui	50.01	3568.55	59.74	97.83	25.67	1596.37	39.95	99.02	32.01	2134.50	46.20	98.70
Persistence	71.21	9708.55	98.53	99.50	34.94	2717.49	52.13	99.86	39.36	3041.56	55.15	99.84
Auth	79.23	11158.89	105.64	96.35	47.34	3857.84	62.11	98.74	39.57	3611.02	60.09	98.82
Recommender	31.22	1258.56	35.48	94.26	24.49	911.24	30.19	95.84	20.27	620.22	24.90	97.17
Image	71.45	8137.23	90.21	98.72	72.48	7328.20	85.60	98.85	42.99	3642.93	60.36	99.43

and discuss its approach to abstracting the target microservice application. Finally, we elaborate on the algorithm that Waterfall uses to obtain the scaling strategy.

3.5.1 Architecture

Figure 3.7 shows the architecture of Waterfall which is based on the MAPE-K control loop [19], [53], [54] with five elements, namely monitor, analysis, plan, execute, and a shared knowledge base.

The Monitor component collects the monitoring data including different metrics from microservices in the target application, aggregates the data, and extracts the features for machine learning performance models and then hands it over to the Analysis component. The Analysis component initializes the microservice graph using the data provided by the Monitor component. Afterwards, It uses the machine learning performance models and scaling algorithm to obtain the new number of replicas for each microservice in the application and sends a change request to Plan Component. The plan component gets the

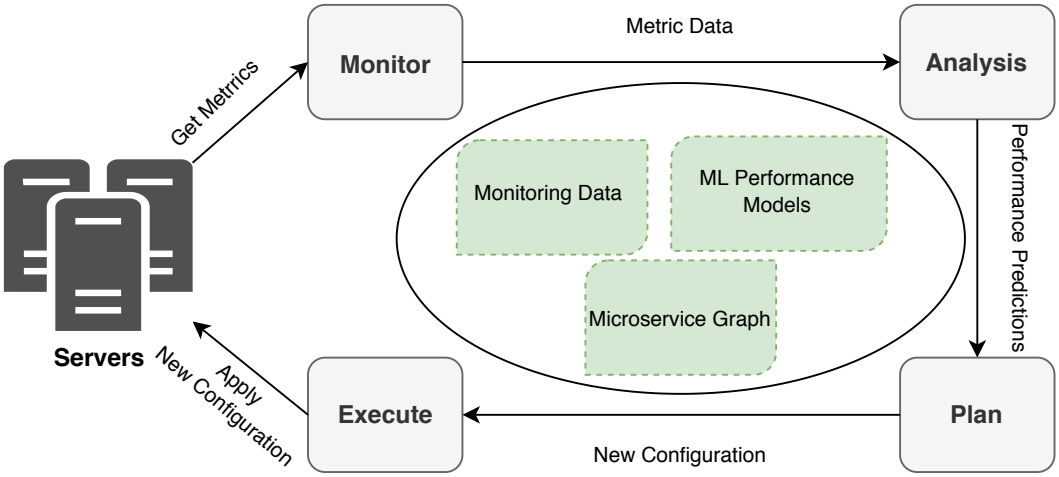


Figure 3.7: Architecture of Waterfall autoscaler.

output of the Analysis component and calculates the new configuration and required changes for the system, and passes the change plan to the Execute component. Execute component interacts with the system and applies the new configuration to the system. The knowledge base contains the microservice graph of the application and machine learning performance models and is shared among the Monitor, Analysis, Plan, and Execute components.

3.5.2 Microservice Graph

Waterfall abstracts the target microservice application as a directed graph, which is called microservice graph, hereafter. In the microservice graph, vertexes represent services, and edges show the dependencies between services. The direction of an edge determines which service sends request to the other one. For instance, consider the following vertex (V) and edge (E) sets for an example microservice graph:

$$\begin{aligned} V &= \{A, B, C\} \\ E &= \{(A, B), (A, C)\} \end{aligned} \tag{3.3}$$

This microservice graph contains three services and two edges. A, B, and C are three different services. The edges (A, B) and (A, C) show that service A calls services B and C respectively. In addition, we assign the following three weights to each directed edge (m, n) between two microservices m and n:

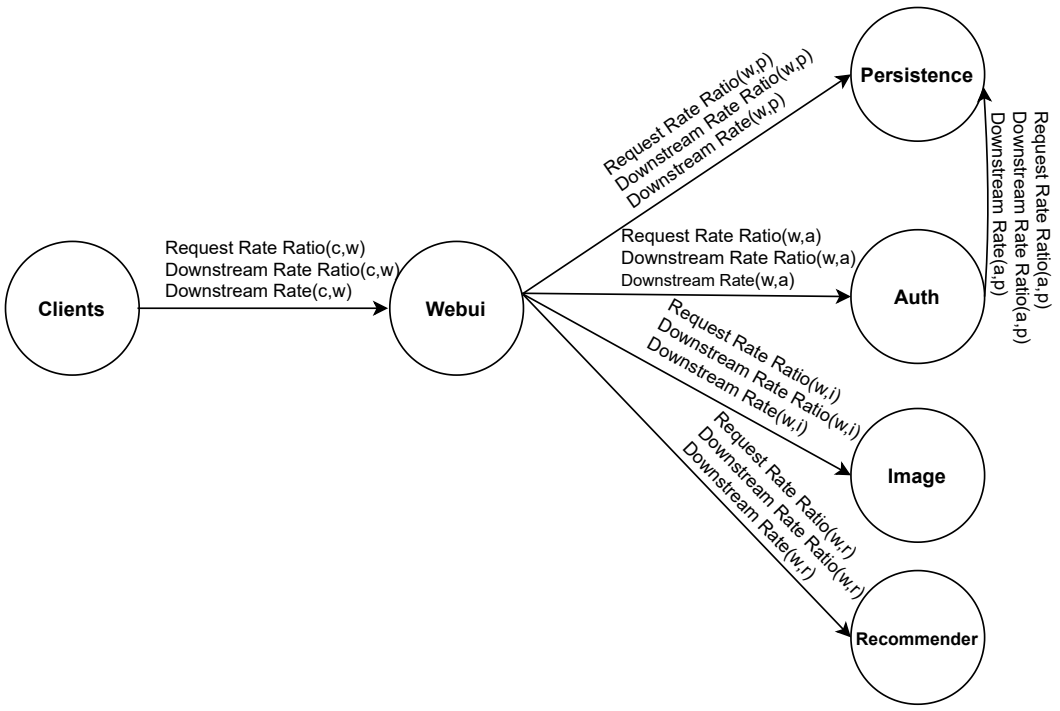


Figure 3.8: Teastore microservice graph.

- **Downstream Rate(m,n)** which is defined in Section 3.2.
- **Request Rate Ratio(m,n)** which is defined for two services m and n as:

$$Request\ Rate\ Ratio(m,n) = \frac{DownstreamRate(m,n)}{RequestRate(n)} \tag{3.4}$$

- **Downstream Rate Ratio(m,n)** which is defined for two services m and n as:

$$Downstream\ Rate\ Ratio(m,n) = \frac{DownstreamRate(m,n)}{RequestRate(m)} \tag{3.5}$$

We calculate these weights for each edge and populate the graph using the monitoring data. Figure 3.8 shows the microservice graph for the Teastore application. The microservice graph for small applications can be derived manually according to service dependencies. There are also tools [61] for extracting the microservice graph automatically.

3.5.3 Scaling Algorithm

Our proposed algorithm for autoscaling of microservices leverages machine learning models to predict the number of required replicas for each service and the impact of scaling a services on the load of other services. This way, we provide a more responsive autoscaler that takes fewer actions to keep the application at the desired performance.

At the end of each monitoring interval, Waterfall initializes the microservice graph weights using monitoring data and runs the scaling algorithm to find the new scaling configuration. The steps in the Waterfall scaling algorithm are summarized in Algorithm 1. The algorithm takes the microservice graph, start node, and monitoring data as input and provides the new scaling configuration as the output. In the beginning, it initializes the New_Config with the current configuration of the system using monitoring data and starts finding the new configuration.

It traverses the microservice graph using the Breadth-First Search (BFS) algorithm and starts the search from the start node. The start node is usually the front-end service, which is the users' interaction point with the application. At each node, the algorithm checks whether the CPU utilization of the service is above or below the target threshold.

In case that the CPU utilization is higher than the threshold, it calls the *scaleOut* function. This function increases the service replicas and predicts the new request rate of the service using Request Model. After predicting the new request rate, it uses CPU Model to predict the new CPU utilization with the new number of replicas and the new request rate. If the new predicted CPU utilization is below the threshold, it considers the new replica as the new configuration for the service. Afterwards, it updates the microservice request rate using the *updateReqRate* function. As Algorithm 3 indicates, function *updateReqRate* updates the *DownstreamRate* value on all edges ending to this microservice based on the *Request Rate Ratio* value on each edge.

If the CPU utilization is less than the threshold, it calls the *scaleIn* function. This function reduces the number of service replicas and predicts the new

request rate of the service using Request Model. It then feeds the new request rate and new replica to CPU Model to predict the new CPU utilization. If the new CPU utilization is still below the threshold, it considers the new replica as the new configuration for service and updates the microservice request rate using the *updateReqRate* function. Otherwise, it keeps the current replica as the configuration of the service.

If the node that is being processed has any children, the algorithm goes to the next step which is applying the effect of change in service replica number on downstream services by calling the *updateDownstreamRate* function. As Algorithm 3 shows, this function updates the *DownstreamRate* value on all edges starting from the current node and ending at child nodes based on the *Downstream Rate Ratio* value on each edge.

After this step, the algorithm continues the BFS search by the next node and repeats the steps mentioned above. After searching the whole graph and inferring the new configuration for each service, the search is over and the algorithm returns the new scaling configuration.

As can be seen in lines 8-11, if the request rate of the service in the current node has been changed in the graph in previous steps, the CPU utilization in the monitoring data is not valid anymore, and we should estimate the new CPU utilization using CPU Model. The *getRequestRate* function calculates the request rate of a node by summing the *Downstream Rate* value on all edges ending to this node.

In cases that the microservice graph of the target application includes a cycle, we need to add a stop condition to the scaling algorithm. For example, this stop condition can be based on a threshold for changes in predicted values or can be achieved simply by ensuring that each node is visited only once in the search process.

3.6 Experimental Evaluation

In this section, we evaluate the performance of Waterfall autoscaler by comparing Waterfall with Kubernetes Horizontal Pod Autoscaler (HPA), which

Algorithm 1: Autoscaling Algorithm

Input: Microservice Graph G, Start Node S, Monitoring Data M
Output: New Scaling Configuration New_Config

```
1  New_Config ⟵ initilize with current config
2  queue ⟵ []
3  queue.append(S)
4  while queue is not empty do
5      service ⟵ queue.pop(0)
6      req_rate_updated ⟵ False
7      req_rate ⟵ getReqRate(G, service)
8      if M[service]['Req_Rate'] == req_rate then
9          cpu_util ⟵ M[service]['CPU_Util']
10     else
11         cpu_util ⟵
             CPU_Model(service, new_config[service], req_rate)
12     curr_req_rate ⟵ req_rate
13     curr_cpu_util ⟵ cpu_util
14     curr_replica ⟵ new_config[service]
15     if cpu_util >= THRESH then
16         (new_replica, pred_req_rate) ⟵
             scaleOut(curr_replica, curr_cpu_util, curr_req_rate)
17         updateReqRate(G, service, pred_req_rate)
18         new_config[service] ⟵ new_replica
19         req_rate_updated ⟵ True
20     else if cpu_util < THRESH ∧ curr_replica > 1 then
21         (new_replica, pred_req_rate) ⟵
             scaleIn(curr_replica, curr_cpu_util, curr_req_rate)
22         if new_replica ≠ curr_replica then
23             updateReqRate(G, service, pred_req_rate)
24             new_config[service] ⟵ new_replica
25             req_rate_updated ⟵ True
26     if G[service].hasChild() ∧ req_rate_updated then
27         updateDownstreamRate(G, service, pred_req_rate)
28     for each v ∈ G[service].adjacent() do
29         queue.append(v)
```

Algorithm 2: Scale Out and Scale In Functions

```
1  Function scaleOut(curr_replica, curr_cpu_util, curr_req_rate):
2      new_replica ⟵ curr_replica
3      pred_cpu_util ⟵ curr_cpu_util
4      while pred_cpu_util > THRESH do
5          new_replica ⟵ new_replica + 1
6          pred_req_rate ⟵ Request_Model(service, curr_replica,
             curr_cpu_util, curr_req_rate, new_replica)
7          pred_cpu_util ⟵ CPU_Model(service, new_replica,
             pred_req_rate)
8      return (new_replica, pred_req_rate)
9  Function scaleIn(curr_replica, curr_cpu_util, curr_req_rate):
10     new_replica ⟵ curr_replica
11     pred_cpu_util ⟵ curr_cpu_util
12     while pred_cpu_util < THRESH do
13         new_replica ⟵ new_replica − 1
14         pred_req_rate ⟵ Request_Model(service, curr_replica,
             curr_cpu_util, curr_req_rate, new_replica)
15         pred_cpu_util ⟵ CPU_Model(service, new_replica,
             pred_req_rate)
16         if pred_cpu_util < THRESH then
17             new_req_rate ⟵ pred_req_rate
18     return (new_replica + 1, new_req_rate)
```

Algorithm 3: Microservice Graph Helper Functions

```
Function getReqRate(Microservice Graph G, Node service):
    req_rate ⟵ 0
    for each (m, n) ∈ G do
        if n == service then
            req_rate ⟵ req_rate + G[m][n]['DownstreamRate']
    return req_rate
Function updateReqRate(Microservice Graph G, Node service,
  new_req_rate):
    for each (m, n) ∈ G do
        if n == service then
            G[m][n]['DownstreamRate'] ⟵
              new_req_rate * G[m][n]['ReqRateRatio']
Function updateDownstreamRate(Microservice Graph G, Node
  service, new_req_rate):
    for each (m, n) ∈ G do
        if m == service then
            G[m][n]['DownstreamRate'] ⟵
              new_req_rate * G[m][n]['DownstreamRateRatio']
```

is the de facto standard for autoscaling in the industry. First, we elaborate on the details of our experimental setup. After that, we present and discuss our experimental results for the comparison of Waterfall and HPA in terms of different metrics.

3.6.1 Experimental Setup

Microservice Application Deployment

We forked the source code of the Teastore application from GitHub and instrumented all Teastore services with a monitoring agent. Moreover, we improved the startup time of the microservice containers by tuning the performance of the Tomcat web server. After these modifications, we recreated the Docker image of each microservice.

We created a Kubernetes[2] cluster as the container orchestration system with one master node and four worker nodes in the Compute Canada Arbutus Cloud[3]. Each node is a virtual machine with 16 vCPU and 60GB of memory running Ubuntu 18.04 as the operating system. We deployed each microservice in the Teastore application as a Kubernetes deployment exposed by a Kubernetes service. The incoming traffic is distributed in a round-robin fashion between pods that belong to a deployment. We imposed constraints on the amount of resources available to each pod using the resource request and limit mechanism in Kubernetes. The resource request is the amount of resources guaranteed for a pod, and the resource limit is the maximum amount of resources that a pod can have in the cluster. We used the same value for both resource request and limit to decrease the variability in pods' performance. Table 3.5 shows the details of CPU and memory configuration for each pod. We configured the startups, readiness, and liveness probes for each pod to measure the exact number of ready pods at any time in the system and also have a recovery mechanism in place for unhealthy pods. We used the Kubernetes API to query or change the number of pods in a deployment.

[2]Kubernetes: `https://kubernetes.io`
[3]Compute Canada Cloud: `https://computecanada.ca`

Table 3.5: Resource request and limit of Teastore services.

Service Name	CPU	Memory
Webui	1200mCore	512MB
Persistence	900mCore	512MB
Auth	900mCore	512MB
Recommender	800mCore	512MB
Image	1100mCore	512MB

Monitoring and Load Generation

Datadog[4], which is an online monitoring service for cloud applications, is used to monitor each microservice in the monitoring component. We deployed Datadog agents on each node of the Kubernetes cluster and instrumented each microservice with the Datadog tracing library. The following metrics are collected for each service:

- CPU utilization (%): The average CPU utilization of pods that belong to a service deployment in percentage.
- Request rate (per second): The sum of request rates of all pods that belong to a service deployment.
- Downstream rate (per second): The downstream rate of each service deployment to other service deployments.
- Replica: The number of pods in a service deployment.

We used Jmeter[5], an open-source tool for load testing of web applications, to generate an increasing workload with a length of 25 minutes for the Teastore application. This workload is a common browsing workload that represents the behaviour of most users when visiting an online shopping store. It follows a closed workload model and includes actions like visiting the home page, login, adding product to cart, etc. Jmeter acts like users' browsers and sends requests sequentially to the Teastore front-end service using a set of threads.

[4]Datadog: `https://datadoghq.com`
[5]Jmeter: `https://jmeter.apache.org`

The number of threads controls the rate at which Jmeter sends requests to the front-end service. We deployed Jmeter on a stand-alone virtual machine with 16 vCPU and 60GB of memory running Ubuntu 18.04 as the operating system.

3.6.2 Results and Discussion

To compare the behaviour and effectiveness of Waterfall autoscaler with HPA, we applied the increasing workload described in the previous section to the front-end service of the Teastore application for 25 minutes. Figures 3.9-3.13 show the average CPU utilization and replica count for each service in the Teastore application throughout the experiment. The red dashed line in CPU utilization plots denotes the CPU utilization threshold that both autoscalers use as the scaling threshold. The green dashed line in each service's replica count plot shows the ideal replica count for that service at each moment of the experiment. The ideal replica count is the minimum number of replicas for the service which is enough to handle the incoming load and keep the CPU utilization of the service below the threshold. According to Figures 3.9-3.13, HPA scales a service whenever the service's average CPU utilization goes above the scaling threshold. However, Waterfall scales a service in two different situations: 1) the CPU utilization of the service goes beyond the scaling threshold; 2) the predicted CPU utilization for the service exceeds the threshold due to scaling of another service. Therefore, when Waterfall scales a service while its CPU utilization is below the threshold (e.g., around the 6^{th} minute in Figure 3.10), it must be due to the predicted performance degradation of the service as a result of scaling of another service(s).

As Figure 3.9 shows, for the Webui service, both autoscalers increase the replica count when the CPU utilization is above the threshold with some delay compared to the ideal state. According to Figure 3.8, as Webui is the front-end service and no other internal services depend on it, scaling of other services does not affect the performance of the Webui service. Hence, all Waterfall's scaling actions for the Webui service can be attributed to CPU utilization.

As can be seen in Figure 3.10, we observe that Waterfall scales the Per-

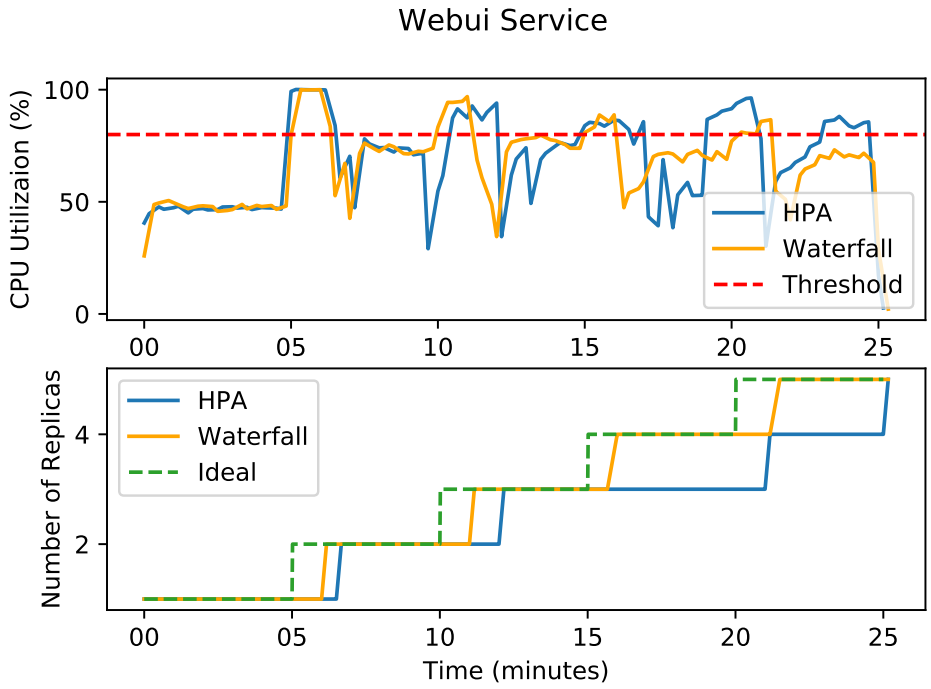


Figure 3.9: The CPU utilization and number of replicas for the Webui service. The red dashed line in the upper plot shows the scaling threshold and the green dashed line in the lower plot denotes the ideal number of replicas throughout the experiment.

sistence service around the 6$^{\text{th}}$ minute, although the CPU utilization is below the threshold. We attribute this scaling action to the decision for scaling the Webui service in the same monitoring interval that leads to an increase in the CPU utilization of Persistence service as Webui service depends on Persistence service. In contrast, as we can see in Figure 3.10, the HPA does not scale the Persistence service at the 6$^{\text{th}}$ minute. Consequently, a short while after the 6$^{\text{th}}$ minute, when the second replica of Webui service completes the startup process and is ready to accept traffic, the CPU utilization of Persistence service increases and goes above the threshold. The other scaling action of Waterfall for Persistence service after the 15$^{\text{th}}$ minute is based on CPU utilization.

Results for the Auth service shown in Figure 3.11 suggest that the increase in the replica count of Auth around the 6$^{\text{th}}$ minute is based on the prediction for the impact of scaling of the Webui service, as the CPU utilization of Auth is below the threshold during this time. On the other hand, we can see that

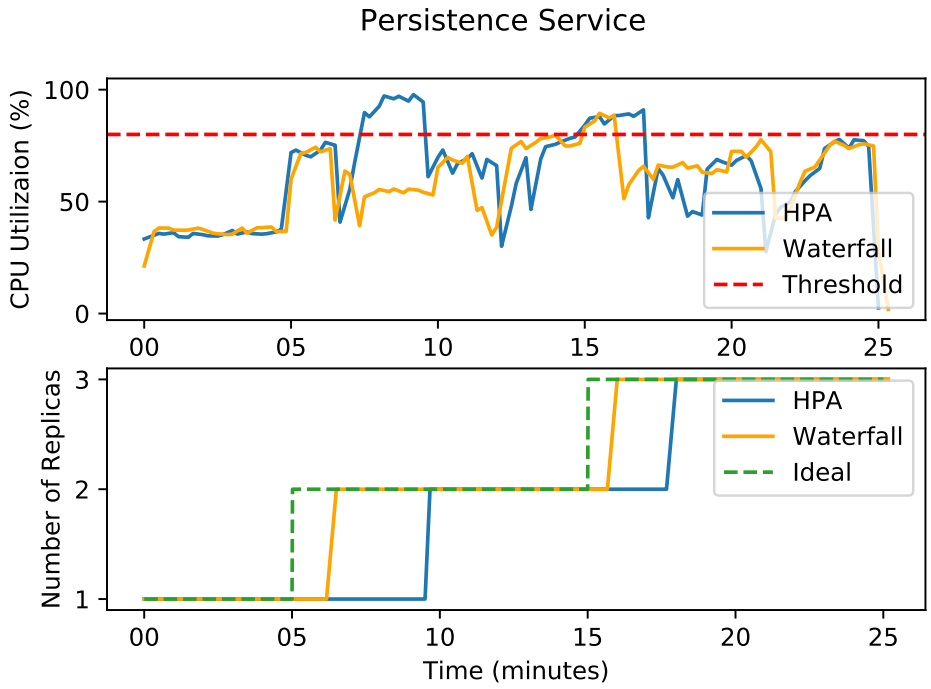


Figure 3.10: The CPU utilization and number of replicas for the Persistence service. The red dashed line in the upper plot shows the scaling threshold and the green dashed line in the lower plot denotes the ideal number of replicas throughout the experiment.

at 6^{th} minute, the HPA does not increase the replica count for Auth service. Therefore, after adding the second replica of Webui, the CPU utilization of Auth reaches the threshold. The other scaling action of Waterfall for Auth after the 20^{th} minute is based on the CPU utilization.

According to the Image service results in Figure 3.12, Waterfall scales the Image service around the 11^{th} minute. This scaling action is due to scaling the Webui service that depends on Image service from two to three replicas in the same monitoring interval. However, HPA does not scale the Image service simultaneously with Webui causing an increase in the CPU utilization of the Image service. For Waterfall, as Figure 3.12 shows, there is a sudden increase in the CPU utilization of Image service right before the time that the second replica of Image service is ready to accept traffic. This sudden increase in CPU utilization of Image service is because of the time difference between the time that Webui and Image services complete the startup process

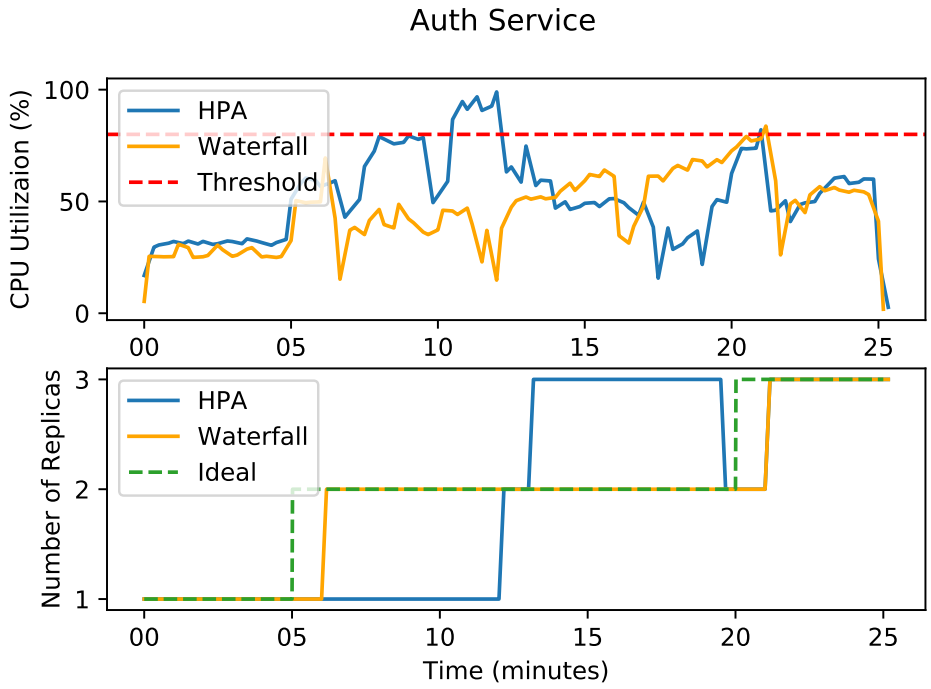


Figure 3.11: CPU utilization and number of replicas for Auth service. The red dashed line in the upper plot shows the scaling threshold and the green dashed line in the lower plot denotes the ideal number of replicas throughout the experiment.

and reach the ready state. During the interval between these two incidents, the Webui service has three replicas; therefore, its downstream rate to Image service increases while the second replica of the Image service is not ready yet.

For the Recommender service, as Figure 3.13 illustrates, during the whole time of the experiment, the CPU utilization is below the threshold. Consequently, there is no scaling action for both autoscalers.

Putting the results of all services together, we can see that the Waterfall autoscaler predicts the effect of scaling a service on downstream services and scale them proactively in one shot if it is necessary. Therefore, it takes fewer actions to maintain the CPU utilization of the application below the threshold. For example, around the 6^{th} minute, we can see from Figures 3.9, 3.10, and 3.11 that Waterfall autoscaler scales the Persistence and Auth services along with Webui in the same monitoring interval. However, HPA scales these services separately in different monitoring intervals.

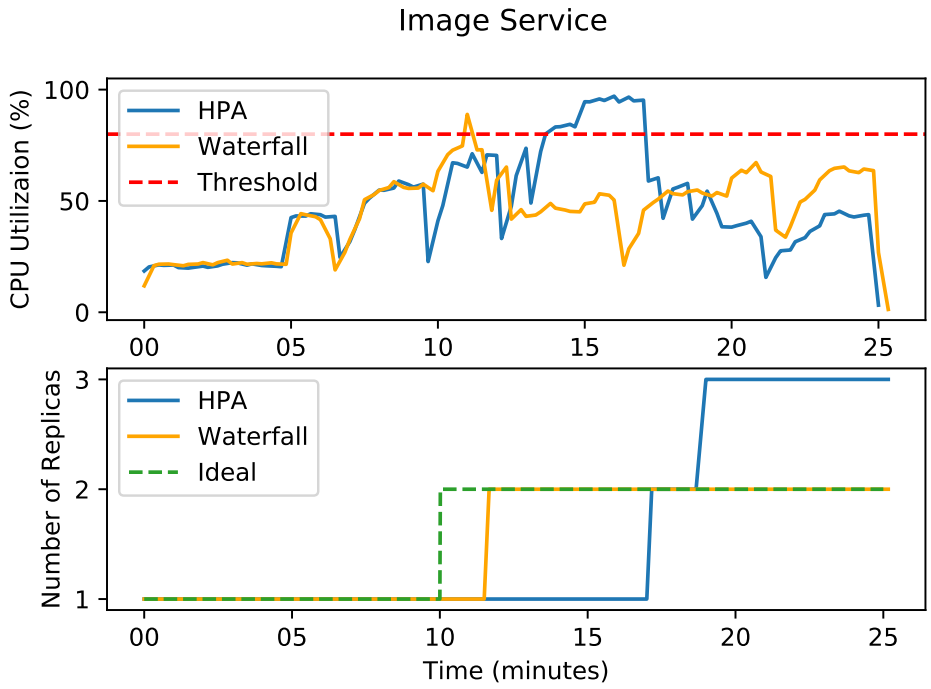


Figure 3.12: The CPU utilization and number of replicas for the Image service. The red dashed line in the upper plot shows the scaling threshold and the green dashed line in the lower plot denotes the ideal number of replicas throughout the experiment.

To quantify the effectiveness of Waterfall compared to HPA, we evaluate both autoscalers in terms of several metrics. Figure 3.14 shows the total number of transactions executed per second (TPS) for Waterfall and HPA throughout the experiment. It can be seen that Waterfall has a higher cumulative TPS than HPA thanks to timely scaling of services.

We repeated the same experiment five times and calculated the average of the total number of served requests, TPS, and response time for both autoscalers over these runs. Table 3.6 shows the results along with the 95% confidence interval. It can be seen that TPS (and the total number of served requests) is 9.57% higher for Waterfall than HPA. The response time for Waterfall is also 8.79% lower than HPA.

Additionally, we have calculated the following metrics for both autoscalers and presented them in Table 3.7:

- CPU>Threshold time: The percentage of time that CPU utilization of

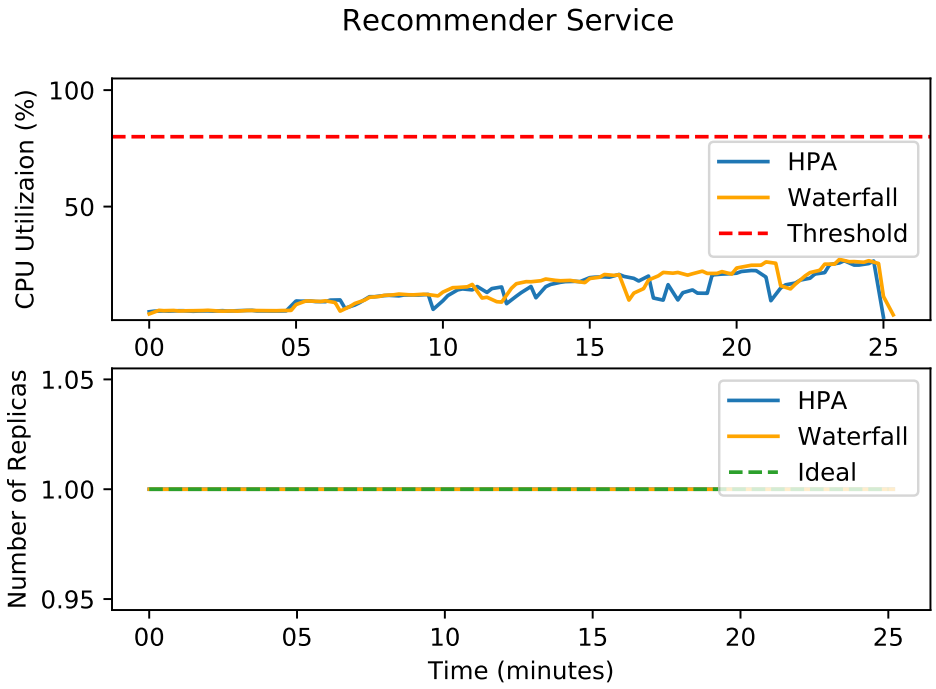


Figure 3.13: The CPU utilization and number of replicas for the Recommender service. The red dashed line in the upper plot shows the scaling threshold and the green dashed line in the lower plot denotes the ideal number of replicas throughout the experiment.

Table 3.6: Comparison of Waterfall and HPA autoscalers in terms of performance metrics.

#	**HPA**	**Waterfall**
Total Request	727270.0 ± 12369.95	796867.4 ± 4594.77
TPS	484.55 ± 8.23	530.93 ± 3.06
Response Time	20.47 ± 0.36	18.67 ± 0.11

the service is above the threshold.

- Underprovision time: The percentage of time that the number of service replicas is less than the ideal state.

- Overprovision time: The percentage of time that the number of service replicas is more than the ideal state.

It can be seen that for all services except the Recommender service, both autoscalers have a nonzero value for CPU>T. However, CPU>T is less for

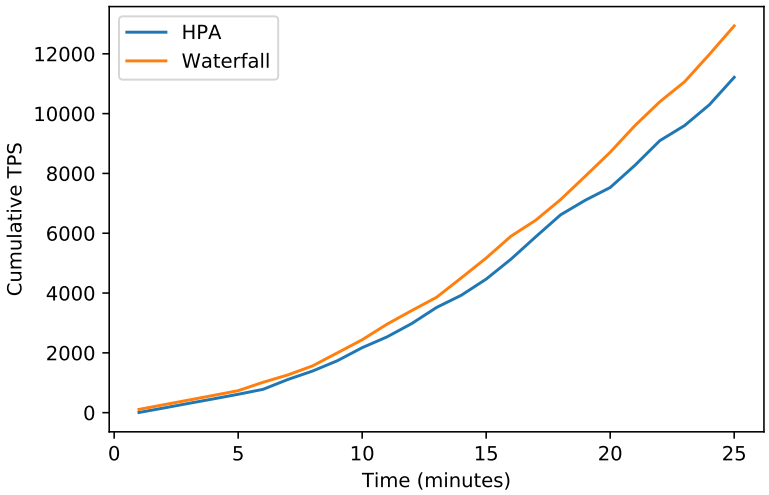


Figure 3.14: Cumulative Transaction Per Second (TPS) of Waterfall and HPA autoscalers.

Table 3.7: Comparison of Waterfall and HPA in terms of CPU>Threshold(T), overprovision, and underprovision time.

Service	**CPU >T**		**Underprovision**		**Overprovision**	
	HPA	Waterfall	HPA	Waterfall	HPA	Waterfall
Webui	∼31%	∼16%	∼54%	∼15.33%	0%	0%
Persistence	∼16%	∼4%	∼28.66%	∼7.33%	0%	0%
Auth	∼6.33%	∼0.33%	∼32%	∼8%	26%	0%
Image	∼13.33%	∼0.33%	∼28%	∼6%	24%	0%
Recommender	0%	0%	0%	%0	0%	0%

Waterfall in all services. Moreover, Waterfall yields a lower underprovision time and zero overprovision time for all services. Despite the overprovisioning of HPA for two services, we observe that Waterfall still provides a higher TPS and better response time; we attribute this to the timely and effective scaling of services by the Waterfall autoscaler.

3.7 Conclusion

We introduced Waterfall, a machine learning-based autoscaler for microservice applications. While numerous autoscalers consider different microservices in

an application independent of each other, Waterfall takes into account that scaling a service might have an impact on other services and can even shift the bottleneck from the current service to downstream services. Predicting this impact and taking the proper action in a timely manner could improve the application performance as we corroborated in this study. Our evaluation results show the efficacy and applicability of our approach.

Waterfall can be adopted as a part of container orchestration platforms to improve the performance of microservice applications in cloud environments. As the future work, we plan to explore the feasibility of adding vertical scaling to the Waterfall autoscaling approach.

Chapter 4

Maintaining the Performance of Containerized Cloud Applications Through Multi-versioning

Containerization technology is widely adopted for the development of cloud applications as it makes the deployment and management of applications in the cloud platform easier. The common practice for maintaining the performance of containerized cloud applications is increasing the application resources by horizontal or vertical scaling. However, this approach may not be cost-effective and applicable in situations where the available resources or budget is limited.

In this chapter, we study an alternative, more cost-effective approach for satisfying the performance requirements of containerized cloud applications. In particular, we investigate how we can satisfy such requirements by applying software multi-versioning to the resource-intensive containers in cloud applications. We demonstrate the efficacy of multi-versioning for satisfying the performance requirements of containerized cloud applications through experiments on the Teastore, a microservice reference test application, and Znn, a containerized news portal application. The results presented in this chapter have been published in ICPE 2020 conference [35].

4.1 Introduction

Maintaining the performance of cloud applications is a challenging task due to uncertainty in the incoming workload. For example, one common problem that may happen for an online web application is the Slashdot effect. The Slashdot effect is a sudden increase in website traffic that occurs when a high-traffic website posts a link to a low-traffic website [3], [4]. If the low-traffic website cannot handle the sudden increase in traffic, it may experience prolonged response times or unavailability. One way to mitigate this situation is to increase the available resources for processing requests by horizontal or vertical scaling. However, this approach can become very expensive and could add high over-provisioning costs, which not every project can afford. An alternative solution could be to have different versions of the services provided by the website. For instance, if the website had lightweight versions of some of its essential, resource-intensive components, it could use them during the high load to reduce its resource usage while maintaining a reasonable response time. A similar example of this *software multi-versioning* concept has been used by the Gmail service, which has a lightweight HTML-based version that is used when the user's browser does not support the feature-rich but resource-heavy JavaScript-based version [21]. By falling back on the lightweight version, the user would still be able to use Gmail with a minimal and simple user interface.

Software multi-versioning has been traditionally applied to mission-critical systems, such as flight or nuclear power plant control systems, to improve their reliability and safety [14], [15], [30], [52]. These systems usually have a monolithic architecture. Therefore, applying multi-versioning to these systems means developing and maintaining multiple versions of the whole system at the same time. This makes multi-versioning an impractical technique and a costly process for non-critical systems. However, the advent of containerization technology, cloud computing, and loosely coupled architectures, such as microservice, has paved the way for employing multi-versioning in non-critical systems. We can apply software multi-versioning to a specific component of a system running inside a container rather than the whole system.

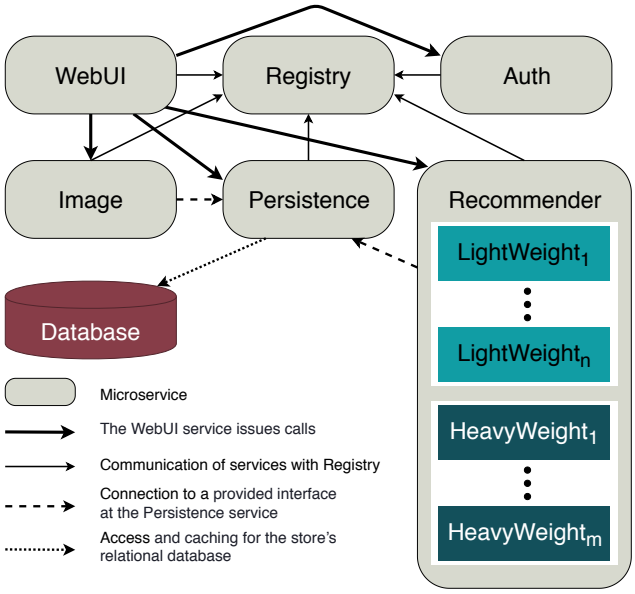


Figure 4.1: High-level architecture of the Teastore application with multi-versioning applied to the Recommender microservice.

In this chapter, we examine how software multi-versioning can help satisfy the performance requirements of containerized cloud applications. We conduct a set of experiments on the performance of two containerized cloud applications under varying loads. In the first experiment, we study the Teastore [55], which is a reference microservices application for benchmarking and performance testing. We apply multi-versioning to this application by developing two versions of its Recommender service. One version is resource-intensive but provides more accurate results. The other version is lightweight but returns less accurate results. In the second experiment, we study Znn [23], a three-tier online news application where we develop two versions of its content-providing component. One version offers high-fidelity content, and the other one provides low-fidelity content.

The rest of the chapter is organized as follows. Section 4.2 presents a motivational example for our approach. In Section 4.3, we present our approach for employing multi-versioning in containerized cloud applications. In Section 4.4, we explain our experimental setup. Section 4.5 discusses the results of our experiments. Section 4.6 gives an overview of the related work. Finally, Section 4.7 concludes the chapter.

4.2 Motivating Example

Consider a newly founded startup that sells its products through an online shopping store. This startup has limited financial resources and would like to run its business in a cost-effective manner. The product manager of this startup has recently decided to add a product recommendation feature to the online store and assigned a developer to this task. The developer implemented this new feature, but in the testing phase, they noticed that this new feature is resource-hungry and becomes a bottleneck in high loads. The developer suggested two possible options for solving this issue: 1) Assign more resources to the recommendation component 2) Use a lightweight algorithm for the recommendation component that generates less accurate recommendations. The product manager rejected the first option and asked the developer to work on a solution that keeps the balance between performance and accuracy based on website load. Therefore, the developer decided to implement two versions of the recommendation feature: resource-intensive version and lightweight version. The resource-intensive version is used when the website traffic is not high, and the lightweight version is used when the website is experiencing a high load, especially during sales events. The website switches between these two versions to continue serving customers with less accurate recommendations during high loads and with more accurate recommendations when the website is not under a heavy load.

4.3 Multi-Versioning in Containerized Cloud Applications

Software multi-versioning is about developing and deploying multiple versions of a software system or a software system component to improve the quality attributes of that software system. We aim to employ this technique for improving the performance of containerized cloud applications.

In general, when we deploy a containerized cloud application, each component resides inside a container created from an image. For components

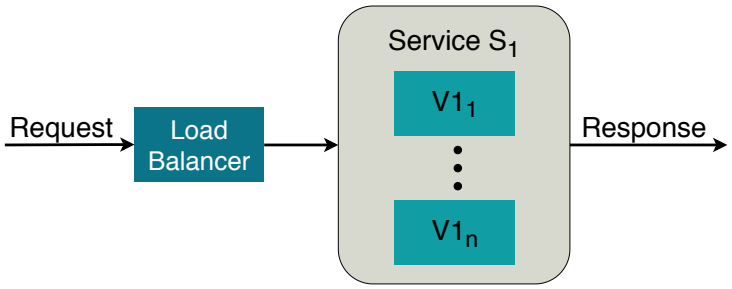


Figure 4.2: High-level architecture of a regular Docker service with round-robin load distribution.

with more than one replica, the incoming requests are distributed equally between replicas. For example, Figure 4.2 shows an example of a Docker service containing n replicas. As shown in Figure 4.2, Docker has an internal load balancing mechanism that distributes the incoming load between replicas in a round-robin fashion.

In our approach, we apply multi-versioning to components of a cloud application to maintain the performance at the desired level. Therefore, we deploy different versions of the same component instantiated from different images. Different versions perform the same operation at different service fidelity. For instance, they may use different algorithms to carry out the same task (similar to our motivating example in Section 4.2). To maintain the performance requirement of the application, we need to distribute the requests between different versions of a component based on a metric, such as response time, CPU utilization, etc., representing the current load of the applications. To this end, we developed an adaptive load balancer that distributes the incoming load based on a set of rules. The rule-set determines how the incoming requests should be distributed between different versions based on the value of the monitored metric. We used a customized version of NGINX[1] to implement this load balancer. Listing 4.1 shows the format of the rules for the load balancer.

The parameters in the rule in Listing 4.1 are as follows:

- `$METRIC`: The metric that is used to check whether a rule should fire.

[1] `https://www.nginx.com`

Listing 4.1: Format of the rules for the load balancer

```
$METRIC $OPERATOR $THRESHOLD ,
    (version $VERSION_NAME perc=$PERCENTAGE;)+
```

Listing 4.2: Example rule for the load balancer

```
RT > 0.4 ,
    version recommender:HeavyWeight perc=40;
    version recommender:LightWeight perc=60;
```

Currently, only `RT` (median response time) is supported.

- `$OPERATOR`: The relational operator ($<, <=, >, >=$ *or* $==$) that is used in the condition to check whether a rule should fire.
- `$THRESHOLD`: The threshold for the metric that is used in the condition to check whether a rule should fire.
- `$VERSION_NAME`: The name of one of the versions of the service.
- `$PERCENTAGE`: The percentage of requests to be directed to the container (between 1 and 100).[2]

Listing 4.2 shows an example rule, in which 40% of the requests are routed to the first version (e.g., resource-intensive version) and the other 60% are routed to the second version (e.g., lightWeight version). We monitor the NGINX's log file and calculate the median response time every five seconds. The value of median response time determines which rule should be used for distributing the load between different versions. NGINX saves the `$time_local`, and `$request_time` for each of the incoming requests. The `$time_local` returns the local time of the machine, and we use that time to identify the requests which were received in the last n seconds. The `$request_time` is the elapsed time since the first bytes were read from the client.

One could argue that software multi-versioning could easily be implemented using `if`-statements inside a component source code. However, source code-

[2]NGINX does not accept 0 as the percentage of requests.

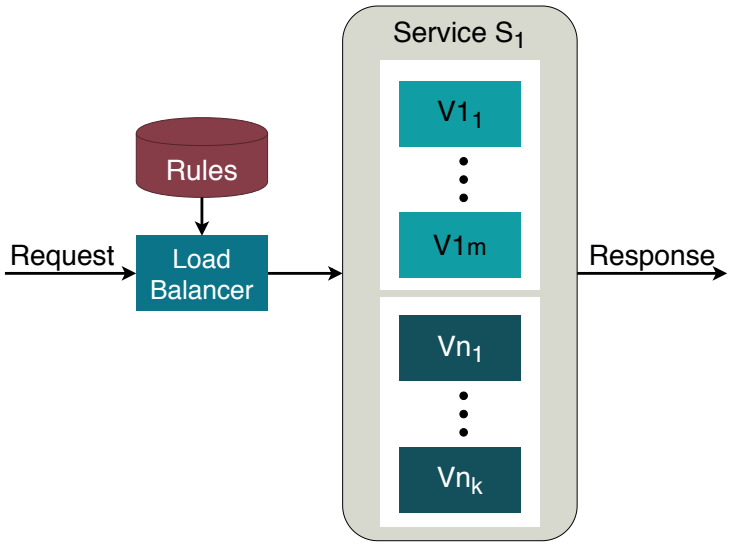


Figure 4.3: High-level architecture of a service with multi-versioning where requests are balanced based on a rule-set.

based solutions are against the separation of concerns principle and clutter the source code. In addition, the maintenance and understanding of source code become challenging. Therefore, our approach provides multi-versioning in containerized cloud applications in a non-cluttered manner.

4.4 Experimental Setup

In this section, we elaborate on our experimental setup. The goal of our experiments is to study the effectiveness of software multi-versioning for satisfying the performance requirements of containerized cloud applications.

4.4.1 Subject Cloud Applications

In our experiments, we use two well-known cloud applications: Teastore [55] and Znn.

Teastore is a reference microservice application developed for performance testing and benchmarking. As Figure 4.1 shows, this application is composed of six microservices and a database emulating an online shopping store. Each microservice and also the database component is encapsulated in a container. We introduced multi-versioning in the Teastore application by developing two

different versions of the Recommender service. This service uses a collaborative filtering algorithm to provide product recommendations and has a parameter that adjusts the retraining frequency. In the lightweight version, this parameter is set so that retraining happens only once at startup time. In the heavyweight version, this parameter is set so that retraining happens multiple times. The source code of the Teastore [3] is publicly available.

Znn [23] is a three-tier web-based application developed for testing and benchmarking of self-adaptive applications. The Znn application emulates a news portal and contains a pool of web servers, a MySQL database with news-related text and multimedia contents, and a load balancer that receives requests from clients and distributes them among the web servers in a round-robin manner. The high-level architecture of the Znn application is shown in Figure 4.4. The original version of the Znn application is not containerized. We containerized the Znn application by encapsulating Web server and database components in two separate containers. Normally, the Znn servers respond to an HTTP request with an HTML page containing a some news with multimedia content such as image and video. To introduce multi-versioning in Znn, we developed two versions of the web server component. The lightweight version returns only news text without any multimedia content to be able to handle a large number of requests. The heavyweight, on the other hand, returns news along with multimedia content. Figure 4.5 shows the high-level architecture of the containerized version of the Znn application with multi-versioning. The source code of the Znn [4] is publicly available.

4.4.2 Experiments

We conducted three experiments for each of the target cloud applications:

- **Heavyweight-only experiment**: In this experiment, all requests are served using the heavyweight versions of the services to provide the best user experience. Therefore, for the Teastore application, all requests are served by the heavyweight version of the Recommender service (constant

[3] `https://github.com/DescartesResearch/TeaStore`
[4] `https://github.com/cmu-able/znn`

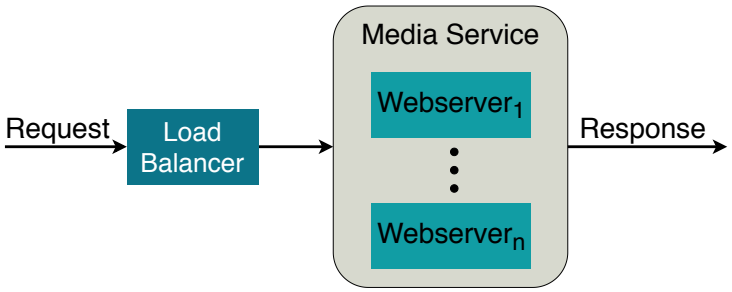


Figure 4.4: High-level architecture of the Znn application

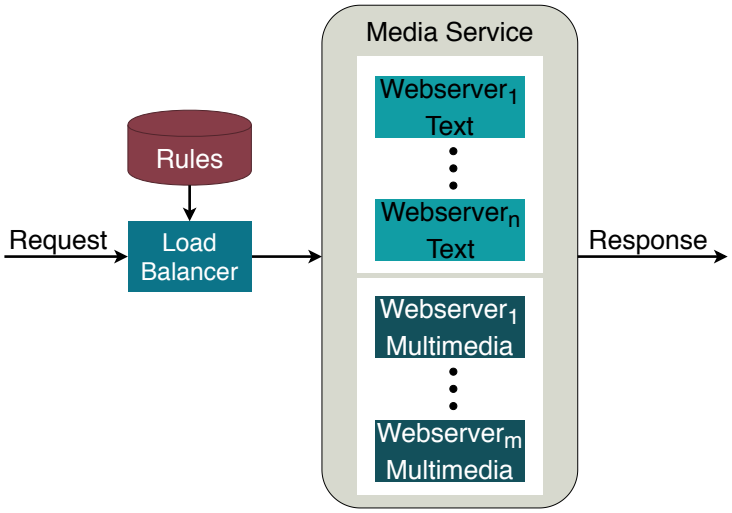


Figure 4.5: Containerized deployment of Znn with two different versions of the `Media` service.

retraining), and for the Znn application, all requests are served by the heavyweight version of the web server component (text + multimedia).

- **Lightweight-only experiment**: In this experiment, we studied the worst-case scenario from users' perspective as all requests are served by the lightweight versions of the services. Hence, for the Teastore application, all requests are served by the lightweight version of the Recommender service (single training), and for the Znn application, all requests are served by the lightweight version of the web server component (only text).

- **Adaptive experiment**: In this experiment, we tried an adaptive approach that falls between two other experiments. The goal is to maintain the performance by distributing the requests between heavyweight and

Table 4.1: A description of the experiments that we conducted for the Teastore and Znn applications

Experiment	Teastore Description	Znn Description
Heavyweight-only	Recommender with multiple training	Multimedia responses only
Lightweight-only	Recommender with single training	Text responses only
Adaptive	Adaptive load distribution	Adaptive load distribution

lightweight versions based on a predefined rule set and the current load of the system. Therefore, for the Teastore application, both versions of the Recommender service, and for the Znn application, both versions of the web server component are used.

These experiments are summarized in Table 4.1. In all experiments, we considered an upper threshold of 450 milliseconds for the median response time of the Teastore application and 1 second for the median response time of the Znn application as the SLA requirement.

4.4.3 Workload

We used Apache JMeter,[5] a tool for load testing of web applications, to generate two workloads for our experiments.

For the Teastore application, we modified the JMeter test plan[6] provided by the Teastore developers and added more items to the shopping cart to put more pressure on the Recommender service. We generated a workload of 100 users who concurrently send HTTP requests to the Teastore application for different purposes, such as visiting the home page, logging in, or adding items to the cart. For each request, users receive an HTML page as the response, and as soon they get the response, they send the next request. The length of this workload is 1,000 seconds.

For the Znn application, we generated an increasing workload simulating multiple users sending requests to the Znn application concurrently. Figure 4.6 shows the shape of the workload. The length of the workload is 2 hours, and

[5] `https://jmeter.apache.org`

[6] The workload's JMeter test plan is available on the project's GitHub repository [36].

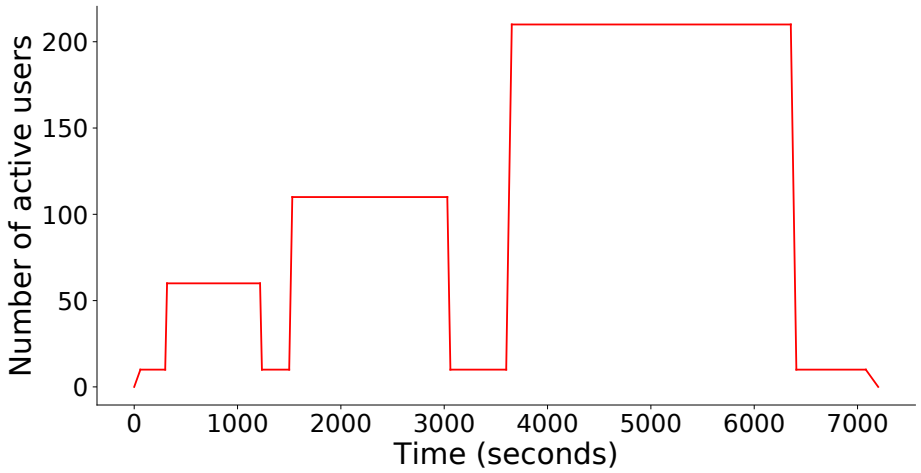


Figure 4.6: Shape of the Znn application workload

Table 4.2: Description of the virtual machines

Cloud	Instance	VCPUs	Memory	OS
Cybera	Experiment	4	8GB	Ubuntu-18.04
Compute Canada	JMeter	4	15GB	Ubuntu-18.04

the number of users increases from 60 to 210 over time. At the highest peak, the number of active users is 210, which means 210 threads concurrently send HTTP requests to the servers.

4.4.4 Deployment and Load Balancing Rule Sets

We provisioned one virtual machine in the Compute Canada cloud[7] and one virtual machine in the Cybera Rapid Access Cloud[8] to deploy our containers for our experiments. In particular, we ran the JMeter script on the Compute Canada cloud and the subject systems on the Cybera cloud. Table 4.2 summarizes the configurations of our virtual machines.

Table 4.3 shows the description of the containers that we used for the experiments. We limited the containers' memory, swap memory, and CPU to prevent them from consuming all of the resources on the host machine. These

[7] `https://www.computecanada.ca/research-portal/national-services/compute-canada-cloud`

[8] `https://www.cybera.ca/services/rapid-access-cloud`

Table 4.3: Description of the containers in the experiments

Name	Docker Image	Memory	Swap Memory	CPU
HeavyWeightRecommende	sgholami/teastore-recommender:HeavyWeight	1G	1G	0.4
LightWeightRecommender	sgholami/teastore-recommender:LightWeight	1G	1G	0.4
Multimedia	alirezagoli/znn-multimedia:v1	1G	1G	0.4
Text	alirezagoli/znn-text:v1	1G	1G	0.4
NGINX	sgholami/nginx-monitoring	unlimited	unlimited	unlimited
NGINX_official	NGINX	unlimited	unlimited	unlimited
MySQL	alirezagoli/znn-mysql:v1	unlimited	unlimited	unlimited

Listing 4.3: NGINX rule set for the Teastore application

```
RT < 0.1 ,
    version recommender:HeavyWeight perc=99;
    version recommender:LightWeight perc=1;
RT < 0.25 ,
    version recommender:HeavyWeight perc=90;
    version recommender:LightWeight perc=10;
RT < 0.4 ,
    version recommender:HeavyWeight perc=80;
    version recommender:LightWeight perc=20;
RT >= 0.4 ,
    version recommender:HeavyWeight perc=70;
    version recommender:LightWeight perc=30;
```

limits were defined based on our experience with the subject systems.

For the Teastore application, we define the rule set presented in Listing 4.3 in NGINX to balance the load between the two versions of the Recommender service. Listing 4.4 shows the rule set for the Znn application. Both rule sets were defined empirically based on observations during preliminary runs of the experiments.

4.5 Experimental Evaluation

In this section, we discuss the results of our experiments for each subject cloud application.

Listing 4.4: NGINX rule set for the Znn application

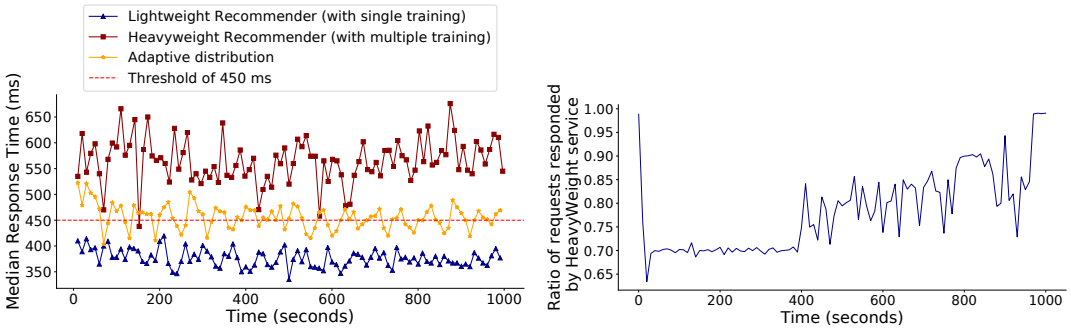
```
RT < 0.1 ,
    version znn-multimedia:v1 perc=99;
    version znn-text:v1 perc=1;
RT < 0.2 ,
    version znn-multimedia:v1 perc=80;
    version znn-text:v1 perc=20;
RT < 0.3 ,
    version znn-multimedia:v1 perc=70;
    version znn-text:v1 perc=30;
RT < 0.6 ,
    version znn-multimedia:v1 perc=40;
    version znn-text:v1 perc=60;
RT < 0.8 ,
    version znn-multimedia:v1 perc=30;
    version znn-text:v1 perc=70;
RT >= 0.8 ,
    version znn-multimedia:v1 perc=20;
    version znn-text:v1 perc=80;
```

4.5.1 Experiments with the Teastore Application

Figure 4.7a shows the median response times of the Teastore application in our experiments. As it can be seen, the median response time in the lightweight-only experiment stays below the threshold and in the heavyweight-only experiment goes above the threshold. For the adaptive experiment, we observe that the median response time fluctuates around the threshold since the load balancer distributes the load between the heavyweight and lightweight versions of the service. In addition, Figure 4.7b shows the ratio of requests that were responded to by the heavyWeight version of the Recommender service in the adaptive experiment.

4.5.2 Experiments with the Znn Application

Figure 4.8a shows the median response time of the Znn application in the heavyweight-only experiment. During this experiment, although all requests were served by the web server with multimedia content, the median response time of the application went far beyond the threshold (up to around 25 seconds)



(a) Median response time of the Teastore application

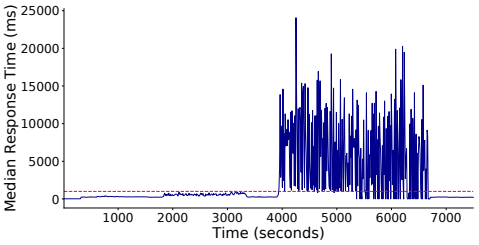
(b) The ratio of requests responded by the HeavyWeight version of the service

Figure 4.7: The Teastore application experiments and the distribution ratio of traffic using software multi-versioning and adaptive load balancing

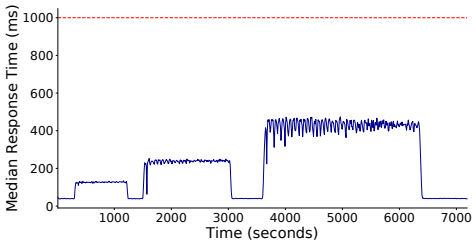
which is not acceptable for users. The high median response time in this experiment indicates that available resources are insufficient for maintaining the performance at the desired level. Figure 4.8b shows the median response time of the Znn application in the lightweight-only experiment. The results suggest that that the available resources are enough for handling the incoming traffic load. However, the user experience is poor since all requests are handled by the text-only version of the web server. Figure 4.9a shows the median response time of the Znn application in the adaptive experiment, where we leverage multi-versioning to distribute the requests between multimedia and text-only versions of the web server. As it can be seen, the median response time is maintained around the threshold and does not go beyond four seconds. In addition, Figure 4.9b shows the ratio of the requests which were responded to by the multimedia web server in the adaptive experiment. We can observe that the system deals with the increases in workload by routing most of the requests (first approximately 50-70% and then approximately 80%) to the text-only version of the web server.

4.6 Related Work

Software multi-versioning is a known technique in software engineering derived from a similar method in hardware systems where multiple redundant hardware modules are used to make a system fault-tolerant [69]. In the hardware

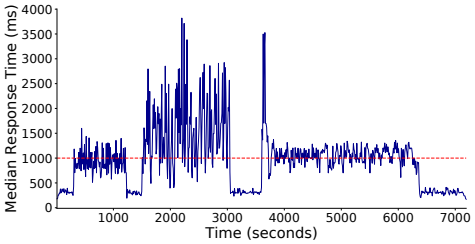


(a) The median response time when running only the multimedia-version of the service

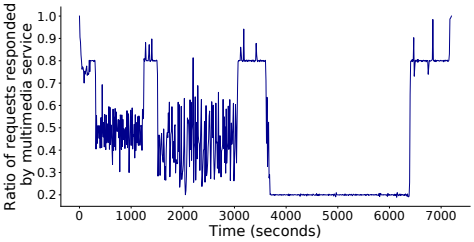


(b) The median response time when running only the text version of the service

Figure 4.8: The Znn application experiments using only the multimedia vs only the text version of the service. Note that the scales on the y-axes are different.



(a) The median response time when running the services with multi-versioning and using the rules defined in Listing 4.4



(b) The ratio of requests responded by the multimedia version of the service

Figure 4.9: The Znn application experiment using software multi-versioning and adaptive load balancing

domain, the redundant modules are usually exactly the same. However, in the software domain, although different versions are functionally equivalent, they are implemented diversely according to the target goal. For example, different versions may use different algorithms, programming languages, compilers, etc. software multi-versioning has been used to improve software systems security [17], [34], [59], [66], safety [44], reliability [25], and availability [42].

In the security domain, the software multi-versioning has been used to detect and mitigate cyber security attacks. Borck et al. [17] developed a system named FEVIS for detecting cyber attacks using multi-versioning. FEVIS generates multiple versions of a program from functionally equivalent code clones and runs them in parallel. It detects attacks through divergence in behaviour

of different versions. Larsenet et al. [59] studied the types of cyber attacks that can be mitigated using diverse versions of a software system and also different approaches for diversifying a software system. Franz [34] introduced a compiler-based method for generating unique versions of a software system as a defence mechanism against security attacks. The authors argued that distributing several versions of a software system decreases the success rate of attackers because they can target only a specific version of the software with each attack. In addition, they need to design a new attack for every version which is challenging and costly. Persaud et al. [66] combined different implementation of SSL library to generate variant versions of a program which is less vulnerable to security attacks.

Software multi-versioning also has been used as an effective strategy for mitigating the design faults and improving the reliability of software systems. Eckhardt et al. [30] investigated the effectiveness of multi-versioning in promoting reliability by developing and comparing the probability of failure in twenty versions of an aerospace application. Each version was developed by an independent programming team and evaluated by developers who were not involved in the development process. Gorbenko et al. [42] proposed an approach for building fault-tolerance service-oriented systems using redundant versions of existing web-services. Wang et al. [76] introduced the idea of applying multi-versioning to critical components of cloud applications to enhance the reliability of cloud applications. They used sensitivity analysis to identify the critical components of a cloud application. Zheng et al. [77] used multi-versioning for improving the reliability of service-oriented systems. They modelled the selection of fault tolerance strategy according to a set of constraints, such as cost, response time, etc., as an optimization problem and solved it using a heuristic algorithm.

4.7 Conclusion

Traditionally, software multi-versioning has been applied only to mission-critical systems due to the high cost of developing and maintaining multiple

versions of the software. However, due to the emergence of containers, cloud computing, and loosely-coupled architecture, multi-versioning is now a feasible technique for non-critical systems.

In this chapter, we studied how software multi-versioning can be used as a technique to satisfy the performance requirements of containerized cloud applications while maintaining the service fidelity as high as possible. We showed the effectiveness of this technique by applying multi-versioning to two example open-source cloud applications and conducting an extensive set of experiments.

As future work, we plan to study to investigate the effectiveness of software multi-versioning for satisfying other non-functional requirements of a containerized cloud applications.

Chapter 5

Conclusion

Cloud applications are becoming increasingly popular thanks to the benefits of cloud computing. One of the vital concepts around cloud applications and cloud services is quality of service. In this thesis, we focused on performance, one of the important dimensions of QoS, and explored three approaches for improving the performance of cloud applications.

In chapter 2 (published in ICPE 2020), we considered a document processing system with monolithic architecture running on a single virtual machine in the cloud and migrated this application to a scalable and performant serverless architecture on the Google Cloud Platform. As results showed, this migration led to a tremendous speed-up in the document processing task with a marginal increase in cost.

In chapter 3 (accepted in CLOSER 2021), we introduced a new approach for autoscaling of cloud applications with microservice architecture. In this approach, we leveraged machine learning to capture the performance behavior of each microservice in applications and also the impact of services on one another. This new machine learning-based approach for the autoscaling of microservice applications outperformed the Kubernetes Horizontal Pod Autoscaler (HPA) in terms of response time and throughput.

In chapter 4 (published in ICPE 2020), we leveraged software multi-versioning, a known concept in software engineering, as a cost-effective way to improve the performance of cloud applications. Evaluation of this approach on two different applications verified the applicability and effectiveness of this approach

for improving the performance of cloud applications.

In future work, we plan to explore the following paths:

- Deploy and compare the performance of the document processing system introduced in chapter 2 on other serverless platforms such as Amazon AWS Lambda, Microsoft Azure Function, and IBM Openwhisk
- Evaluate the feasibility of adding vertical scaling to the autoscaler introduced in chapter 3 to provide more fine-grained scaling and improve the resource utilization
- Investigate the effectiveness of software multi-versioning for satisfying other non-functional requirements of cloud applications such as reliability, security, etc.

References

[1] M. L. Abbott and M. T. Fisher, *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[2] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari, and A. Erradi, "Predictive autoscaling of microservices hosted in fog microdata center," *IEEE Systems Journal*, 2020.

[3] S. Adler, *Addendum to the slashdot effect internet paper*, 1999.

[4] ——, "The slashdot effect: An analysis of three internet publications," *Linux Gazette*, vol. 38, no. 2, 1999.

[5] G. Adzic and R. Chatley, "Serverless computing: Economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 884–889.

[6] Amazon. (2019). "Amazon aws lambda," [Online]. Available: `https://aws.amazon.com/lambda/` (visited on 12/23/2019).

[7] ——, (2019). "Amazon fargate," [Online]. Available: `https://aws.amazon.com/fargate/` (visited on 12/23/2019).

[8] ——, (2019). "Aws lambda costumer case study," [Online]. Available: `https://aws.amazon.com/lambda/resources/customer-case-studies/` (visited on 12/23/2019).

[9] ——, *Amazon ec2 spot instances*, `https://aws.amazon.com/ec2/spot/`, Accessed: 2020-10-25, 2020.

[10] ——, *Aws auto scaling*, `https://aws.amazon.com/types-of-cloud-computing/`, Accessed: 2020-12-25, 2020.

[11] ——, *Aws auto scaling*, `https://aws.amazon.com/autoscaling/`, Accessed: 2020-10-25, 2020.

[12] Apache. (2019). "Openwhisk," [Online]. Available: `https://openwhisk.apache.org/` (visited on 12/23/2019).

[13] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: Modeling techniques and their applications," *Journal of Internet Services and Applications*, vol. 5, no. 1, p. 11, 2014.

[14] A. Avizienis and J. P. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, no. 8, pp. 67–80, 1984.

[15] A. Avizienis and J.-C. Laprie, "Dependable computing: From concepts to design diversity," *Proceedings of the IEEE*, vol. 74, no. 5, pp. 629–638, 1986.

[16] Belval *et al.*, *A python module that wraps the pdftoppm utility to convert pdf to pil image object*, `https://github.com/Belval/pdf2image`, 2020.

[17] H. Borck, M. Boddy, I. J. De Silva, S. Harp, K. Hoyme, S. Johnston, A. Schwerdfeger, and M. Southern, "Frankencode: Creating diverse programs using code clones," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 604–608.

[18] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems*, Springer, 2009, pp. 48–70.

[20] P. Calçado, "Building products at soundcloud—part i: Dealing with the monolith," *Retrieved from: https://developers. soundcloud. com/blog/building-products-at-soundcloud-part-1-dealing-withthe-monolith*, 2014, Accessed: 2020-10-25.

[21] G. H. Center, *Gmail help*, `https://support.google.com/mail/answer/15049?hl=en`, Accessed: 2019-09-27.

[22] T. Chen and R. Bahsoon, "Self-adaptive trade-off decision making for autoscaling cloud-based services," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 618–632, 2015.

[23] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the rainbow self-adaptive system," in *Proceedings of the 9th Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, IEEE, 2009, pp. 132–141.

[24] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015.

[25] C. Cigsar and Y. Lim, "Modeling and analysis of cluster of failures in redundant systems," in *Proceedings of the 2nd International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2017, pp. 119–124.

[26] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 393–403.

[27] G. Cloud. (2019). "Cloud functions," [Online]. Available: `https://cloud.google.com/functions/` (visited on 12/23/2019).

[28] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4195–4202, 2020.

[29] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and ulterior software engineering*, Springer, 2017, pp. 195–216.

[30] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692–702, 1991.

[31] T. Erl, R. Puttini, and Z. Mahmood, *Cloud computing: concepts, technology, & architecture*. Pearson Education, 2013.

[32] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *2014 IEEE International Conference on Cloud Engineering*, IEEE, 2014, pp. 195–204.

[33] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "Above the clouds: A berkeley view of cloud computing," *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

[34] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the New Security Paradigms Workshop*, ACM, 2010, pp. 7–16.

[35] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei, "A framework for satisfying the performance requirements of containerized software systems through multi-versioning," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 150–160.

[36] ——, *DockerMV*, `https://github.com/pacslab/DockerMV`, Accessed: 2019-10-19.

[37] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2019, pp. 1994–2004.

[38] A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, and T. Dauphinee, "Migrating from monolithic to serverless: A fintech case study," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 20–25.

[39] A. Goli, N. Mahmoudi, H. Khazaei, and O. Ardakanian, "A holistic machine learning-based autoscaling approach for microservice applications."

[40] Google. (2019). "Google cloud run," [Online]. Available: `https://cloud.google.com/run/` (visited on 12/23/2019).

[41] ——, (2019). "What is serverless?" [Online]. Available: `https://cloud.google.com/serverless-options` (visited on 01/27/2020).

[42] A. Gorbenko, V. Kharchenko, and A. Romanovsky, "Using inherent service redundancy and diversity to ensure web services dependability," in *Methods, Models and Tools for Fault Tolerance*, Springer, 2009, pp. 324–341.

[43] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 157–167.

[44] E. Gracie, A. Hayek, and J. Borcsok, "Evaluation of fpga design tools for safety systems with on-chip redundancy referring to the standard iec 61508," in *Proceedings of the 2nd International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2017, pp. 386–390.

[45] A. W. Harley, A. Ufkes, and K. G. Derpanis, "Evaluation of deep convolutional nets for document image classification and retrieval," in *International Conference on Document Analysis and Recognition (ICDAR)*.

[46] J. M. Hellerstein *et al.*, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[47] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[48] S. Ihde and K. Parikh, "From a monolith to microservices + rest: The evolution of linkedin's service architecture," *Retrieved from: https://www.infoq.com/presentations/linkedin-microservices-urn/*, 2015, Accessed: 2020-10-25.

[49] W. Iqbal, M. N. Dailey, and D. Carrera, "Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications," *IEEE Systems Journal*, vol. 10, no. 4, pp. 1435–1446, 2015.

[50] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 25–32.

[51] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[52] J. P. J. Kelly, T. I. McVittie, and W. I. Yamamoto, "Implementing design diversity to achieve fault tolerance," *IEEE Software*, vol. 8, no. 4, pp. 61–71, 1991.

[53] J. Kephart, J. Kephart, D. Chess, C. Boutilier, R. Das, J. O. Kephart, and W. E. Walsh, "An architectural blueprint for autonomic computing," *IBM White paper*, pp. 2–10, 2003.

[54] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[55] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2018, pp. 223–236.

[56] Kubernetes, *Kubernetes hpa*, `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`, Accessed: 2020-10-25, 2020.

[57] M. S. Kurz, "Distributed double machine learning with a serverless architecture," *arXiv preprint arXiv:2101.04025*, 2021.

[58] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2019, pp. 80–90.

[59] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 276–291.

[60] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[61] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using service dependency graph to analyze and test microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 2, 2018, pp. 81–86.

[62] Martin Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[63] T. Mauro, "Adopting microservices at netflix: Lessons for architectural design," *Retrieved from https://www. nginx. com/blog/microservices-at-netflix-architectural-best-practices*, 2015, Accessed: 2020-10-25.

[64] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2017, pp. 405–410.

[65] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture.*" O'Reilly Media, Inc.", 2016.

[66] B. Persaud, B. Obada-Obieh, N. Mansourzadeh, A. Moni, and A. Somayaji, "Frankenssl: Recombining cryptographic libraries for software diversity," in *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*, 2016, pp. 19–25.

[67] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[68] R. Smith, "An overview of the tesseract ocr engine," in *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, IEEE, vol. 2, 2007, pp. 629–633.

[69] I. Sommerville, *Software Engineering GE.* Pearson Australia Pty Limited, 2016.

[70] J. Spillner, Y. Bogado, W. Benitez, and F. López Pires, "Co-transformation to cloud-native applications: Development experiences and experimental evaluation," in *8th International Conference on Cloud Computing and Services Science (CLOSER), Funchal, Portugal, 19-21 March 2018*, SciTePress, 2018.

[71] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017.

[72] A. K. Talukder, L. Zimmerman, *et al.*, "Cloud economics: Principles, costs, and benefits," in *Cloud computing*, Springer, 2010, pp. 343–360.

[73] L. Toader *et al.*, "Graphless: Toward serverless graph processing," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, 2019, pp. 66–73.

[74] M. Wajahat, A. Karve, A. Kochut, and A. Gandhi, "Mlscale: A machine learning based application-agnostic autoscaler," *Sustainable Computing: Informatics and Systems*, vol. 22, pp. 287–299, 2019.

[75] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019, pp. 1288–1296.

[76] L. Wang and K. S. Trivedi, "Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[77] Z. Zheng, M. R. T. Lyu, and H. Wang, "Service fault tolerance for highly reliable service-oriented systems: An overview," *Science China Information Sciences*, vol. 58, no. 5, pp. 1–12, 2015.

[78] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.