Strategies For Building Performant Containerized Applications

by

Mikael Sabuhi

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

Doctor of Philosophy

in

Software Engineering and Intelligent Systems

Department of Electrical And Computer Engineering
University of Alberta

Abstract

The evolution of cloud computing in the last decade has offered unprecedented access to sizable, configurable computing resources with minimal management effort. Containerization of applications, particularly through Docker, has been pivotal in this progression. As modern software increasingly relies on various cloud services, designing performant cloud applications has emerged as a critical concern. Key attributes of such applications include reliability, scalability, efficiency, fault tolerance, and responsiveness. This thesis seeks to address the challenges intrinsic to creating performant cloud applications by developing strategies aimed at achieving these characteristics through: 1) the application of autoscaling techniques to enhance scalability, efficiency, and responsiveness; 2) the introduction of a methodology for assessing the impact of Docker image upgrades on containerized applications to prevent performance degradation; and 3) the utilization of microservices architecture to develop scalable, reliable, and fault-tolerant cloud applications.

In our initial research, we propose a pioneering approach to optimize the performance and resource usage of containerized cloud applications using adaptive controllers grounded in control theory. Our methodology harnesses the capacity of neural networks to capture the intrinsic non-linearity of these applications, and adapts the parameters of a proportional-integral-derivative (PID) controller to accommodate environmental changes. The outcomes demonstrate significant enhancements in resource utilization and a reduction in service level agreement violations, surpassing the performance of other examined autoscaling techniques.

In the subsequent study, we present a method to evaluate the performance implica-

tions of Docker image upgrades on cloud software systems and their correlation with application dependencies. Our case study of 90 official WordPress images underscores the need for comprehensive performance testing before upgrades, the importance of maintaining a performance repository for reporting test results, and the potential benefits of extending semantic versioning to encompass performance modifications. This investigation encourages an enlightened approach to Docker image management, promoting enhanced cloud application performance.

Lastly, we introduce Micro-FL, a fault-tolerant federated learning framework crafted to enhance the reliability and scalability of cloud-based machine learning platforms. By incorporating a microservices-based architecture within Docker containers, Micro-FL overcomes challenges typically associated with federated learning, such as resource constraints, scalability, and system faults. Performance assessments demonstrate Micro-FL's capability to efficiently manage faults and streamline federated learning processes, offering a more robust and scalable solution for federated learning.

The research work presented in this thesis provides deep insights, actionable recommendations, and effective and thoroughly evaluated approaches for building performant cloud applications.

Preface

The research presented in this thesis was conducted at the ENergy digiTizAtIon Lab (ENTAIL) and the Analytics of Software, Games, and Repository Data (ASGAARD) Lab, under the guidance of Dr. Petr Musilek and Dr. Cor-Paul Bezemer. This thesis is predicated on three principal publications:

1. M. Sabuhi, N. Mahmoudi, and H. Khazaei, "Optimizing the Performance of Containerized Cloud Software Systems Using Adaptive PID Controllers," ACM Trans. Auton. Adapt. Syst., vol. 15, no. 3, Art. no. 8, Sep. 2020.

2. M. Sabuhi, P. Musilek, and C.-P. Bezemer, "Studying the Performance Risks of Upgrading Docker Hub Images: A Case Study of WordPress," in Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), pp. 97-104, 2022.

3. M. Sabuhi, P. Musilek, and C.-P. Bezemer, "Micro-FL: A Fault-Tolerant Scalable Microservice Based Federated Learning Platform," IEEE Internet of Things Journal, 2023, under review.

Throughout my graduate studies, I had the privilege of collaborating with Dr. Qing Zhao's team from the University of Alberta on the NSERC/Honeywell CRD Project - Data Analytics and Learning-Based Industrial Diagnosis and Monitoring System. This three-year project focused on exploring novel machine learning techniques for anomaly detection. A notable outcome of this collaboration, which is not incorporated in this thesis, is as follows:

1. M. Sabuhi, M. Zhou, C.-P. Bezemer, and P. Musilek, "Applications of Generative Adversarial Networks in Anomaly Detection: A Systematic Literature Review," in IEEE Access, vol. 9, pp. 161003-161029, 2021.

Chapter 2 is grounded on our Transactions paper [196], wherein I was accountable for the design of a machine learning model for performance modelling of containerized cloud applications and the creation of a control theoretical auto-scaling method to optimize the application's performance. Nima Mahmoudi provided valuable assistance in refining the idea, developing the machine learning model, and deploying the containerized application. Dr. Hamzeh Khazaei served as the supervisory author and participated in concept formation. Chapter 3 draws from our conference paper [198]. In this research study, I took charge of designing the research methodology, developing the web crawler, and conducting the performance test on the cloud environment. Dr. Petr Musilek and Dr. Cor-Paul Bezemer, acting as supervisory authors, offered significant help with concept formation and manuscript editing. Chapter 4 is predicated on a journal paper [197] currently under review for the IEEE Internet of Things Journal. My responsibilities in this research study encompassed the system design of the federated learning platform, the construction of Docker images, preparation of the cloud environment for tests, and performance analysis of the federated learning platform. Dr. Cor-Paul Bezemer and Dr. Petr Musilek, as supervisory authors, contributed to concept formulation and manuscript editing. Additionally, in our review paper on anomaly detection using Generative Adversarial Networks (GANs) [199], I was tasked with orchestrating the review steps to conduct a systematic literature review on the applications of GANs for anomaly detection. Ming (Chloe) Zhou assisted me with the literature review process. Moreover, Dr. Cor-Paul Bezemer and Dr. Petr Musilek, as supervisory authors, offered valuable insights and inputs to enhance the quality of the review paper, remaining involved throughout all stages of the review process.

Acknowledgements

I would like to acknowledge the invaluable guidance and support provided by my supervisors, Dr. Petr Musilek and Dr. Cor-Paul Bezemer, throughout my PhD studies. Their expertise and encouragement have been instrumental in the successful completion of this research. Furthermore, I am sincerely appreciative of Dr. Qing Zhao, whose priceless insights have greatly enriched our collaborative endeavors on the Honeywell project.

I am grateful for the support received during the course of this research. I would like to express my appreciation to Honeywell International Incorporation for their partial support. Furthermore, I extend my gratitude to Google Cloud for their contributions through the Google Cloud Research Scholarships and provision of cloud computing resources. I am also thankful to Cybera for their assistance in providing cloud computing resources.

I would also like to extend my gratitude to my friends in the ENTAIL and ASGAARD Lab for their emotional and mental support, which has been crucial in navigating the challenges of this journey.

Finally, I would like to express my deepest appreciation to my family for their consistent support and encouragement at every stage of my life. Their belief in me has been a constant source of motivation, and I am truly grateful for their presence in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Emerging from the amalgamation of concepts such as heterogeneous distributed computing, grid computing, utility computing, and autonomic computing, cloud computing has presented itself as a remarkably transformative paradigm [202]. With myriad definitions offered for cloud computing [17, 159, 228], it is the National Institute of Standards and Technology that provides an apt explanation: cloud computing represents a model that promotes ubiquitous and convenient access to a vast pool of configurable computing resources such as networks, servers, storage, applications, and services, which can be provisioned swiftly with minimum management or service provider intervention [159]. Due to its inherent attributes, cloud computing has attracted increasing interest within both industry and academia. This interest is driven by its potential to transform software system operation models, thereby promising high system availability, scalability, and performance [79]. Modern cloud-based applications are typically composed of a diverse range of services and applications, each with unique resource requirements to fulfil their Service Level Agreement (SLA) and maintain a high Quality of Service (QoS). In this context, building a cloud application that can deliver high-quality service without infringing upon the SLAs becomes an imperative. In other words, creating an application that is not only performant but also able to maintain this performance under varying loads and conditions, forms the cornerstone of this endeavor.

1.1 Performant Cloud Software Systems

The notion of a performant cloud software system refers to a cloud-based platform that delivers services or capabilities with high efficiency and performance. Key characteristics of such a system include reliability, scalability, efficiency, fault tolerance, and responsiveness. *Reliability* refers to a system's ability to consistently deliver expected outcomes over a specified period in a particular environment [189]. This means the software should perform as expected at all times, even in the face of various failures such as software, hardware, service failures, or power outages. *Scalability* reflects a system's ability to expand in response to increased demand while maintaining its performance objectives or SLAs [5]. In the context of cloud applications, this means being capable of automatically allocating additional computing resources to handle increased demand or workload. Techniques for achieving this include scaling up (adding resources to an existing machine), scaling out (adding more machines), or autoscaling (dynamically adjusting resources based on real-time demand). *Efficiency* is determined by how effectively a software system utilizes its allocated resources such as CPU, memory, and network capabilities. One approach to enhance the efficiency of cloud software systems is through resource allocation, a process that involves distributing available resources to cloud applications over the internet while considering the available infrastructure, SLAs, cost, and energy factors [3]. *Fault tolerance* is a system's ability to continue operations despite the failure of certain components. The system should be capable of detecting, identifying, and recovering from faults. Techniques for designing fault-tolerant cloud software systems include component redundancy, tolerance policies, and load balancing fault tolerance [55, 169]. *Responsiveness* refers to how efficiently and effectively a cloud software system responds to user requests. This implies that the software's execution time should be optimized to reduce latency and improve operation speed.

1.2 Containerization

Virtualization, a method that allows the sharing of physical resources among different users and applications, is seen as one of the foundational concepts behind the advent of modern cloud environments [29]. By consolidating numerous underutilized machines into a single system, virtualization can optimize resources in data centers, including hardware, software, energy, and monetary investments. It also improves process reliability and fault tolerance through performance isolation, process migration, and replication across geographically disparate locations [241].

An alternative approach to conventional virtualization, known as *container-based virtualization*, *Operating System (OS)-virtualization*, or *containerization*, is increasingly being recognized as a viable option. This approach significantly reduces resource overhead, thereby enhancing the efficiency of data center utilization [200]. In the containerization scheme, distinct instances known as containers operate atop a shared OS kernel, with necessary isolation measures in place. A key distinction to note is that compared to hypervisor-based virtualization, which uses virtual machines (VMs), the virtualized objects in containerization are largely confined to global kernel resources [209]. This particular characteristic allows containerization to operate multiple virtual environments on a shared host kernel, utilizing fewer CPU, memory, and networking resources. Consequently, containerization can provide more efficient, scalable, and cost-effective resource management solutions in cloud infrastructures, thereby explaining its rising popularity [232]. More specifically, container management is best handled explicitly by a distributed system, rather than leaving it up to the user to initiate a container within each task [249].

Among various containerization solutions available, *Docker* has emerged as a leading choice [252]. As a principal technology in this space, Docker offers a lightweight and minimal-overhead solution for implementing containerized applications, aligning perfectly with the increasingly popular microservice architecture. The rapid startup

times associated with Docker containers make it easy to perform swift scaling actions, both horizontally and vertically, thus outperforming alternatives like virtual machines. The unique characteristics of Docker align perfectly with microservices architecture where each service is typically encapsulated in its own container, thereby promoting isolation, optimized resource utilization, fault-tolerance and improved security.

1.3 Thesis Motivations and Objectives

Designing a performant cloud software system represents a substantial challenge, requiring meticulous consideration to ensure that the system embodies the essential qualities of reliability, scalability, efficiency, fault tolerance, and responsiveness. In this context, containerization emerges as a potent instrument, opening new avenues to enhance these essential attributes and fundamentally redefine the capabilities of cloud software systems. This technology offers unique features that can be harnessed to enhance the performance of cloud software systems. In light of these developments, this thesis seeks to provide strategies for leveraging containerization to construct performant cloud software systems. This thesis endeavors to present strategies for harnessing containerization to build performant cloud software systems. The proposed strategies will primarily focus on improving the scalability, efficiency, responsiveness, reliability, and fault tolerance of these systems.

The central objectives of this thesis are as follows:

- *Objective 1: Investigate strategies that can potentially enhance resource allocation, application responsiveness, and scalability of containerized applications to attain optimal performance.*
  - Evaluate the viability of utilizing Neural Networks (NNs) to construct a dependable performance model, tailored to developing adaptive auto-scalers within a control-theoretical framework.

  - Investigate and categorize the types of controllers capable of effectively preserving desired performance alongside key metrics, such as Service Level Agreement (SLA),in cloud-based software systems.
  - Identify which controller can tackle resource provisioning tasks with the utmost efficiency.

- *Objective 2: Investigate the impact of updating the Docker image of applications on their overall performance.*
  - Investigate the impact of the upgrade of the application on its performance
  - Investigate the correlation between the application's performance and updates made to its dependencies.

- *Objective 3: Investigate the advantages of employing microservices design patterns for creating performant cloud applications.*
  - Investigate methods to migrate from monolithic applications to a microservice-based architecture.
  - Examine the advantages of implementing microservices design in terms of enhancing scalability, reliability, responsiveness, and fault tolerance.

To accomplish our objectives, we undertake three distinct research studies. The first study, which corresponds to Objective 1, encompasses an exhaustive analysis of a variety of autoscaling techniques. Subsequent to this evaluation, we propose an adaptive PID controller aiming for optimizing the performance, improving the responsiveness of the application, optimizing resource utilization, and reducing service level agreement (SLA) violations. The second study, targeting Objective 2, proposes a methodology for assessing the impact of Docker image upgrades on their performance and potential subsequent effects on the user experience (responsiveness). The central focus of this study lies in understanding the implications of Docker image upgrades

from the vantage points of reliability, responsiveness, and resource utilization. The third study, pertaining to Objective 3, investigates a novel type of cloud application integrating machine learning systems - in this case, Federated Learning systems. The study examines the effects of adopting microservices design patterns on the performance of such cloud applications. The primary emphasis of this study is on enhancing the scalability, reliability, fault tolerance, and resource utilization of cloud applications.

The motivations and findings derived from our research studies, which constitute the contributions of this thesis, are summarized as follows:

Research Study 1: "Optimizing the Performance of Containerized Cloud Software Systems using Adaptive PID-Controllers" [196] (Chapter 2)

Motivation: The improvement of scalability, responsiveness, and resource utilization in containerized cloud applications can be achieved through the implementation of autoscaling techniques. Prior research has delved into various autoscaling techniques, encompassing non-adaptive autoscaling methods [23, 72, 78] and adaptive autoscaling methods [60, 123, 151]. A primary challenge associated with non-adaptive techniques lies in the autoscaler's lack of knowledge about the performance of the scaled cloud application. Capturing the behavior of highly non-linear systems such as cloud applications is a demanding task. Therefore, our motivation was to devise an autoscaling method that harnessed the capabilities of neural networks for modeling the performance of the cloud application and control theory for adapting the autoscaler to changes in the cloud environment, thereby optimizing performance, resource utilization, and responsiveness.

Findings: Our results indicate that using neural networks, we could effectively model the performance of the cloud applications. The neural network presented a Mean Squared Error (MSE) of 5,931.20 and 8,248.60, a Mean Absolute Error (MAE) of 42.34 and 71.57, and a Mean Absolute Percentage Error (MAPE) of 5.96% and 11.278% for the training set and the validation set, respectively. By integrating this

performance model with the adaptive PID controller, our findings demonstrated that all controllers were able to maintain the average response time within the desired operating region. Control-theoretical methods responded faster to disturbances and had fewer SLA violations compared to the scaling heat algorithm. Among control-theoretical approaches, the proposed adaptive controllers notably outperformed the non-adaptive counterparts in both SLA and rise time. Furthermore, we observed that adaptive controllers showed enhanced capabilities in handling resource management tasks by significantly improving over-provisioning, efficient provisioning, and under-provisioning.

Research Study 2: "Studying the Performance Risks of Upgrading Docker Hub Images: A Case Study of WordPress" [198] (Chapter 3)

Motivation: Performance has become an increasingly critical aspect of modern software. Some common performance metrics are response time, throughput, and resource usage [33]. Performance testing aims to answer "what if" questions, such as the impact of software configuration changes or workload changes on the system performance [101]. Despite its significance, performance testing is not widely implemented in the industry [33] or by developers [135]. We hypothesized that the same applies to Docker images. While the performance of individual software components within the image may have been tested, their collective performance in operation has likely never been examined. This oversight could have adverse effects on the performance of cloud applications from the perspectives of reliability, responsiveness, and resource utilization. While several studies have analyzed Docker's performance [69, 146, 194, 242], our research is the first to explore the performance impact of upgrading Docker images. Consequently, we were motivated to propose a strategy to better comprehend the performance risks associated with upgrading a Docker image, with particular attention to its dependencies.

Findings: We selected WordPress as the subject for our performance analysis. Our results illustrated that the considerable variation in relative response time improve-

ments for the studied upgrades implies that predicting performance effects based solely on the WordPress version in the image is challenging. This suggests that WordPress's performance is mostly driven by other components in the image. It is also challenging to forecast how upgrading a WordPress image will alter performance, based on its WordPress or Apache version. A major version upgrade of the PHP consistently improved WordPress's performance, indicating that WordPress's performance is highly dependent on the PHP version. Upgrading the last patch version in a minor/major version to the first available patch version in a minor/major version of PHP invariably enhanced WordPress's performance. We propose that Docker Hub should allow users to provide performance measurements of an image and that Semantic versioning should extend to cover performance changes.

Research Study 3: "Micro-FL: A Fault-Tolerant Scalable Microservice Based Federated Learning Platform" [197] (Chapter 4)

Motivation: Cloud computing has gained popularity due to its resource availability and ease of use, offering an excellent platform for deploying and serving machine learning applications in the cloud environment. Federated learning, a machine learning paradigm that supports collaborative model training across different entities while preserving data privacy, provides a promising alternative to traditional machine learning systems. However, current federated ML frameworks encounter challenges, including high hardware requirements, limited scalability, and significant vulnerability to system faults. Aiming to design a performant federated learning systems, we propose utilizing a microservices architecture to address the issues of scalability and fault tolerance in federated learning systems as a containerized cloud application.

Findings: Our research found that using a microservices architecture for federated learning systems can address scalability issues, improving system performance in response to an increase in workload (users) by horizontally scaling the services within the federated learning system. Furthermore, we discovered that employing redundancy techniques could improve the reliability and fault tolerance of existing federated

learning systems, thereby resolving the issue of a single point of failure in these systems.

Together, discussed research studies, provide multiple pathways for enhancing the performance of containerized cloud applications, addressing a variety of challenges associated with the current design and management of these systems. Each study contributes unique insights and proposes practical, implementable strategies for improving the performance and overall quality of cloud applications.

1.4 Thesis Outline

The remaining sections of this thesis are structured as follows. Chapter 2 introduces our autoscaling technique tailored to optimize the performance of cloud applications. Chapter 3 details our study examining the implications of upgrading Docker Hub images, using WordPress as a case study. Chapter 4 introduces our proposed microservices-based platform, designed to offer scalability and fault tolerance for federated learning. The concluding chapter, Chapter 5, encapsulates the key findings and contributions of our research studies, offering insights into potential avenues for future research.

Chapter 2

Optimizing the Performance of Containerized Cloud Software Systems using Adaptive PID-Controllers

2.1 Abstract

Control theory, with its robust mathematical foundation, has demonstrated its efficacy in the design and implementation of controllers. This approach avoids the issues commonly associated with non-control theoretic controllers. State of the art auto-scaling controllers suffer from one or more of the following limitations: 1) lack of a reliable performance model, 2) using a performance model with low scalability, tractability or fidelity, 3) being application or architecture-specific leading to low extendability and 4) no guarantee on their efficiency. Consequently, in this chapter, we strive to mitigate these problems by leveraging an adaptive controller, which is comprising of a neural network as the performance model and a PID controller as the scaling engine. More specifically, we design, implement and analyze different flavours of these adaptive and non-adaptive controllers, compare and contrast them against each other to find the most suitable one for managing containerized cloud software systems at runtime. The controller's objective is to maintain the response time of the controlled software system in a pre-defined range, and meeting the Service Level

Agreements (SLA) while leading to efficient resource provisioning.

2.2 Introduction

Modern distributed systems need to have robust mechanisms for dealing with changes in their performance in order to be responsive and cost-effective at the same time. This is mainly due to the stochastic nature of the underlying infrastructure's performance, variable workload, and possible failures common in the current complex computing systems. Therefore, the modern distributed systems need to have self-adaptive capabilities to sense the changes in the environment and react accordingly. Various methods have been proposed to address and implement this [9, 13, 98, 208, 216], but there has been very limited research on using control-theory-based solutions.

Control theory has been the go-to approach for many adaptable systems, especially in physical systems due to its predictability, mathematical guarantees, and effectiveness. To use control-theory-based approaches in managing the performance of large-scale software systems effectively, we need accurate performance models of the computing software system [25]. However, building models with acceptable accuracy has proven to be lengthy and error-prone for the modern complex distributed systems [24, 96, 111, 112]. This has led to the current ad-hoc control theoretical solutions which are application-specific and not proven to be extendable for other systems [15, 206].

In this work, we plan to leverage neural networks to design an adaptive performance model that can maintain optimized performance for containerized cloud software systems. Our proposed solution leverages the guarantees and robustness associated with control theoretical approaches. We evaluate our approach in various settings for a generic three-tier containerized application that has been deployed on Google Cloud Platform (GCP). The control objective is achieved by scaling the number of containers in the application tier. Moreover, we shed some light on the control theory's applicability in designing performant software systems. More specifically, we address the following research questions:

- **RQ2.1:** *Can we use Neural Networks to build a reliable performance model for designing control-theoretical adaptive auto-scalers?*

  Designing performance models for large-scale distributed systems that maintain an acceptable trade-off between tractability and fidelity is a very challenging task [111, 112]. In this work, we aim to evaluate if Neural Networks can be used to design a tractable performance model with a high degree of fidelity and extendability.

- **RQ2.2:** *Which type of controllers can effectively maintain the performance and metrics of interest, i.e., Service Level Agreement, Mean Squared Error, and Mean Absolute Error in software systems?*

  Several run-time indicators are considered to see which controller is capable of maintaining the system's response time in the desired operating region with fewer violations. We designed and evaluated fixed PI/PID controllers and adaptive PI/PID counterparts to investigate this question. We compare these controllers to find the most suitable one for distributed software systems.

- **RQ2.3:** *Which controller can handle the task of resource provisioning more efficiently?*

  In this case, we strive to find out which controller can make the best use of the available resources (in our case containers). The goal is to use the resources as it is necessary without over/under-provisioning. To this end, the performance of the proposed adaptive PI and PID controllers are compared against their non-adaptive counterparts as well as a pure reactive (Scaling Heat Algorithm [25]) algorithm by considering two different workloads.

The remainder of this chapter is organized as follows. Section 2.3 discusses the background concepts for this research project. Section 2.4 introduces and compares the related work in the literature. Section 2.5 presents our novel control-theory-based methodology. Section 2.6 outlines our evaluation methodology for the approaches

presented in this work.Section 2.6.3 discusses the results obtained in our evaluation. In Section 2.7, we tried to identify the most important threats to our work's validity as well as some avenues for future research. Finally, Section 2.8 concludes the chapter.

2.3 Background

In this section, we briefly introduce the important concepts that are being used in this chapter, namely, container virtualization and control theory.

Docker containers provide us with a lightweight and low overhead solution to implement containerized application and the well known microservice architectures [161]. Their fast startup enables us to perform rapid-scaling actions, both horizontal and vertical, compared to other solutions like virtual machines.

Cluster management platforms like Kubernetes [129], Mesos [162], and Swarm [61] allow the software system to be deployed and reconfigured at large scale. These orchestrators provide the deployment team with the ability to configure the software without worrying about the underlying infrastructure. These platforms also contain simple reactive auto-scaler units for the deployed cloud application out of the box.

As shown in Figure 2.1, in the context of control theory, each control system consists of several blocks, e.g., sensor, actuator, feedback controller, and system under control. We briefly explain each of these elements.

- *Control Objective*: refers to the control system's purpose. For example, it could be controlling the average response time of the controlled system to be less than t seconds or lay in a pre-defined range.

- *Set-Point*: represents the desired value for the output of the controlled system. For instance, the set point for average response time could be 1000 ms. Note that the set point could be a range as well.

- *Sensor*: refers to a component that enables us read the system output, or in our case, the performance variables. For example, average response, fail ratio,

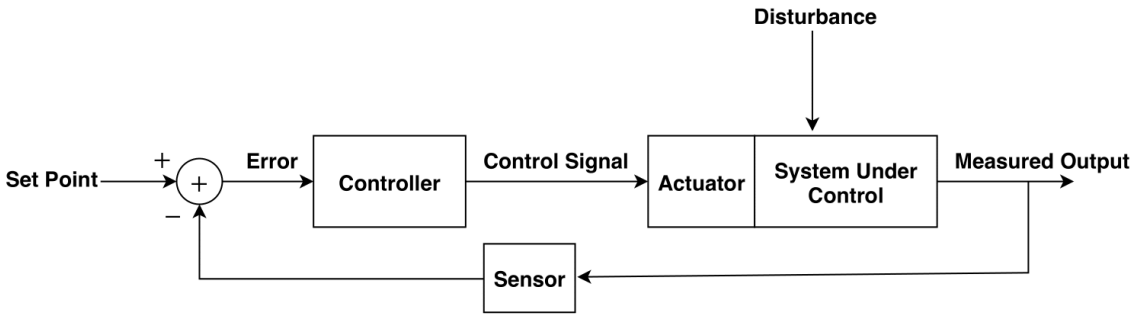


Figure 2.1: Block diagram of a simple control-theoretical feedback loop.

queue length, etc. In fact, it is a software component used for monitoring the controlled system, e.g., Locust and Jmeter.

- *Error*: denotes the difference between the set-point and the measured value of the output read by the sensor, representing the deviation from the desired value for the control objective.

- *Control Signal*: represents the value computed by adopting a specific controller with regards to the error, e.g., having a 400 ms error, the controller might signal for creating two more containers.

- *Controller*: describes a mechanism or algorithm that calculates the control signal to achieve the required control objective, considering the error value. We refer to this algorithm as the control law.

- *Actuator*: refers to a mechanism that can be used to affect the controlled system. For instance, the number of Virtual Machines or containers.

- *System Under Control*: denotes a system to be controlled by adjusting the actuator(s). For example, a containerized cloud application as a system that its control objective is met by creating/removing replicated containers.

The Proportional-Integral-Derivative (PID) controller, also known as "three term" controller [11], still is the most popular controller in the industry due to its simplicity

and transparency. The control signal of the continuous-time PID controller is described as follows; please note that in this study we used discrete-time form of the PID controller (discussed in Section 2.5.2) and this continuous-time form is discussed here only due to its simplicity:

$$u(t) = K_P e(t) + K_I \int_0^t e(t) + K_D \frac{de(t)}{dt} \tag{2.1}$$

Where $u(t)$ is the control signal calculated by the controller, $e(t)$ is the current error, and K_P denotes the proportional gain, K_I the integral gain, and K_D the derivative gain. The purpose of each of these gains are described as follows [11]:

- *Proportional Gain* K_P: shows the sensitivity of the control system to the current error.
- *Integral Gain* K_I: shows the sensitivity of the control system to past errors. Integral term acts as a low-frequency compensator to reduce the steady-state error.
- *Derivative Gain* K_D: shows the sensitivity of the control system to the future trend of the error. Derivative term acts as a high-frequency compensator to improve the transient behaviour of the system.

In order to tune the aforementioned PID controller parameters, interested readers are encouraged to refer to [11] for a thorough overview of PID controllers and their tuning methods.

2.4 Related Work

In this section, we discuss the prior work related to our study of optimizing the performance of containerized software systems using adaptive PID-controllers. In particular, we discuss the related work on performance modelling of cloud software systems, non-adaptive and adaptive control-theoretical methods for auto-scaling.

2.4.1 Cloud Software System Performance modelling

Performance unpredictability and responsive auto-scaling of the cloud applications are listed in the top 10 obstacles for adopting the cloud [17]. Xiong et al. [240] present a novel approach for studying computer service performance in cloud computing. To fulfill the Quality of Service (QoS) guaranteed services in such a computing environment, they seek to find the relationship among the maximum number of users, the minimal service resources, and higher service levels. The authors introduce a queuing network model and then use an approximation method to compute the Laplace transform of a response time distribution. Moreover, they model the Web server and service center as an infinite queue for single-class customers.

Qian et al. [183] propose a hierarchical modelling approach to analytically evaluate Quality of Experience (QoE) of Online Service providers (OSPs) who are using cloud environments. They use four sub-models, namely an outbound bandwidth model, a cloud computing availability model, a latency model, and a cloud computing response time model. These sub-models are combined into one whole model using a redirection strategy graph. Their proposed approach is suitable for endless interactions in this environment. Moreover, one can easily change these identified sub-models and also add other sub-models to the existing model.

Khazaei et al. [116] put forth an analytical model for performance evaluation of cloud server farms. The authors model a cloud server farm as a $M/G/m$ queuing system, considering it a Markov process. Then, they employ embedded Markov chain techniques to analyze the performance of the cloud server farms and verify its accuracy. In their follow-up study [114], they describe a new approximate analytical model based on the Markov chain model for performance evaluation of cloud server farms. The model determines the relationship between the number of servers and input buffer size with performance metrics such as the mean number of tasks in the system, blocking probability, and the probability that a task will obtain immediate

service as well as response time distribution characteristics. In another study [113], the authors introduce a performance model suitable for analyzing large-sized Iaas clouds' service quality, using interacting stochastic models. They leverage both analytical and simulation modelling to address the complexity of cloud computing systems. Their proposed model fuses cloud centers' important characteristics such as batch arrival of user requests, resource visualization, and realistic servicing steps to obtain the important performance indicators such as task blocking probability and total waiting time incurred on user requests. Also, cloud centers' performance with a high degree of virtualization and Poisson batch task arrivals is evaluated in [115]. The proposed model is based on a two-stage approximation technique where first, they model the non-Markovian process with an embedded semi-Markov process, which is then modelled by an embedded Markov process but only at the time instants of super-task arrivals.

Malik et al. [153] provide an in-depth analysis, modelling, and verification of some of the open-source state-of-art VM-based cloud management platforms. They leverage high-level Petri nets (HLPN) to model and assess the software systems' structural and behavioural characteristics with the advantage of providing firm mathematical representations. The authors verified the models using SMT-Lib and Z3 solvers.

Chang et al.[49] highlight the fact that the heterogeneity of the workload in real Iaas Cloud Data Centers (CDCs) makes the performance modelling of complicated Iaas CDCs a challenging task. Their study studies a situation in which the number of virtual CPUs requested by each customer job is different. They present a hierarchical stochastic modelling approach for performance analysis of CDCs to quantify the impact of variation in job arrival rate, buffer size, and the maximum vCPUs numbers on the cloud service quality.

Shekhar et al. [204] present an online data-driven approach that leverages Gaussian Process-Based machine learning techniques to build run-time predictive models of the performance of the system under different interference intensity. This model can

adapt itself to the changes in the workload. The reason for selecting Gaussian Process to model the latency variations due to varying workload is that these types of models require a small number of hyperparameters, and also they can model the non-linear behaviour of the target system. However, these models are probabilistic, and Then they use this model to make run-time decisions for vertical scaling of the resources.

2.4.2 Non-adaptive Control Theoretical Auto-Scaling

Gergin et al. [78] propose a decentralized autonomic architecture based on a fixed PID controller for a n-tier application. They implemented the proposed method on a custom data mining web application based on the FIFA 1998 Workload. For this three-tier application, three separate PID controllers control the number of Virtual Machines (VMs) for each tier to maintain the corresponding CPU utilization at a constant value. Another interesting study Barna et al. [23], evaluates the performance of a non-adaptive PID controller in maintaining the desired behaviour for a web application on SAVI as the private and Amazon EC3 as the public cloud provider. The controller has been tuned manually by trial and error, and the control objective of this controller is to keep the CPU utilization within a specific range.

The problem of control granularity and decoupled control is discussed in [144]. They point out that most of the available cloud controllers function without direct knowledge about the cloud software system behaviour and performance metrics.

Control theory based adaptation using a Fixed PI controller is evaluated and compared with the threshold-based and model-based method in [72]. Introducing smooth and sharp variations in the cloud application workload, these three controllers' performance in maintaining the CPU utilization around 70% is evaluated on a minimal three-tier web application. Looking into results from an efficiency and effectiveness perspective, the PI controller exhibits better resource utilization and faster settling time and rise time.

2.4.3 Adaptive Control Theoretical Auto-Scaling

Several studies on adaptive controllers have been carried out; for example, a vertical elastic controller for memory is presented in [66] along with a method for error smoothing. The control actuator is the memory size allocation, and the target is to maintain the desired response time.

To handle the capacity shortage in cloud infrastructure, the brownout concept is extended from electrical grids to cloud software systems in [123]. During high load, brownout downgrades the user experience, e.g., by decreasing the optional content to be served (i.e., dimmer value); in doing so, the Service Level Objective (SLO) can be maintained. Brownout-compliant applications help support more users and consume less resources while satisfying the SLO. An adaptive PI controller is synthesized for coping with changes in the number of users and the environment. There are several studies on brownout-compliant cloud software systems. For example, Maggio et al. [151] study the applicability of adaptive PI, adaptive deadbeat, adaptive PID, and Feedforward-feedback controllers on Brownout compliant systems. The performance of these controllers against changes in application requirements and resource availability is evaluated. The results of this work indicate that the feedforward-feedback controller has better performance while requiring significant engineering effort. Therefore, since the adaptive PID controller is simpler to implement, it is a preferable choice.

Event-based application brown-out as an improved approach to brown-out is presented in [60] based on the queue-length of pending requests. In this study, improvement in control objectives is reported combining PI controller with machine learning algorithms. Moreover, Nydlander et al. [174] improve this event-based control by proposing a more accurate model of brown-out applications using queuing theory. Quantitative comparison for the proposed cascade controller with original Brown-out and event-based brown-out shows that better performance is achieved by having two control levels: one for the inner-loop and one for the outer-loop. The flexibility of

control theory in dealing with software systems is undeniable; one can find interesting adaptive controllers designed using brown-out for load balancing strategies, e.g. [64, 178].

Baresi et al. [22] enumerate the benefits of containerizing the applications and investigate the performance of auto-scaling in both container and Virtual Machine level. They extend the EcoWare [21] framework to achieve compatibility with containerized applications and then develop an autonomic control theoretical approach for auto-scaling of a cloud application. A new dynamic model for the controlled system is presented to model the application's response time as a function of assigned cores and request rate.

To address the incompatibility of the non-functional software models derived from the architectural description of the software with control theoretical approaches, Arcelli et al. [14] use the Modelica library to represent a Queuing Network. This library provides some adjustable parameters for controlling the behaviour of the cloud software system. Moreover, Model Identification Adaptive Controller(MIAC) based on the layered queuing model and optimal control is presented in [26]. This non-linear model is linearized around the operating point to tune the optimal controller parameter. Moreover, Incerto et al. [95] propose a control algorithm for horizontal and vertical auto-scaling of a cloud software system based on Model Predictive Control(MPC) strategy is presented. In this study, they consider a compact approximate representation of queuing networks based on ordinary differential equations(ODEs) to meet the performance requirements by the model predictive controller. However, the authors point out that the only technical limitation of their proposed method is the single class assumption in the QN model, which they address this limitation in their follow up paper [94].

Several comprehensive surveys on the application of control theoretical methods are found in the recent literature. Ullah et al. [220] look into available methods from both control solution view and elasticity view and review available work on control

theoretical methods for cloud application elasticity and outlines the existing challenges and trends in adopting such methods. Filieri et al. [70] review and elaborate on the current control strategies for self-adaptive software systems starting from goal identification to the verification and validation of the controlled system and at the end, highlight the open challenges, both from the software engineering and the control theory perspective. Moreover, a systematic literature review on control theoretical software adaptation has been presented in [206].

Considering the previous studies on this area, we found that the most challenging problem in adopting control theory in cloud software system adaptation is modelling the software systems. According to previous studies in this area, researchers propose various modelling methods for software systems, such as using queuing networks, to make their auto-scaler adaptive to the environment's changes. However, modelling such highly stochastic and non-linear systems,e.g., cloud applications, is a painstaking and costly task and requires a significant engineering effort. Therefore, in this research project, we propose an approach for data-driven modelling of the software systems leveraging neural networks due to the performance data's abundance. To evaluate our proposed method in **RQ2.1**, we investigate the possibility of using neural networks for reliable performance modelling of cloud applications. Moreover, in **RQ2.2** and **RQ2.3**, we evaluate the performance of the proposed performance modelling method along with an adaptive controller in maintaining the control objectives and from an efficiency point of view.

2.5 Methodology

In this section, we go over the details of the methodology proposed in this research study.

2.5.1 Performance Modelling

The inherent nature of the cloud software systems exhibits some non-linear behaviours at run-time. Modelling such systems is a non-trivial task and sometimes impossible due to the complexity of the non-linear systems. Nevertheless, due to the constant changes in the cloud environment, this model will not accurately represent the system after a while. Linear approximation of these non-linear models only works around the operating point and will not be valid if any significant changes occur in the system's operating point. Considering all these problems, we attempt to develop a new approach for modelling cloud software systems and further adjusting it to its environment changes. Due to the abundance of the data, we propose a data driven approach to obtain a non-linear model of the cloud software system.

The neural networks can be a convenient solution to tackle the problems mentioned above. According to the universal approximation theorem [59, 90], we can use a feed-forward network with a linear output layer and at least one hidden layer with some specific activation functions,e.g. logistic sigmoid functions, to approximate any continuous function provided that the enough hidden layers and neurons are given [81]. Several modelling approaches are developed and validated in [176]. Most of these modelling approaches are performed on the electrical and physical systems. For instance, [6] uses Recurrent High-Order Neural Networks (RHONN) for real-time discrete non-linear modelling of an induction motor. A neural network-based non-linear auto-regressive moving average with exogenous inputs for modelling a piezoelectric actuator has been presented in [54]. These studies motivated us to carry out some experiments to examine neural networks' applicability for modelling cloud software systems. Here, we formalize Hypothesis 1 to obtain the performance model for cloud applications.

Hypothesis 1. In any situation, the performance of a containerized cloud application is a non-linear function of the number of requested containers, the number of successfully

provisioned containers, and the number of users with some time-delays.

$$\begin{aligned}\hat{y}(n+1) =& f(u(n), ..., u(n-n_u), \\ & z(n), ..., z(n-n_z),\, d(n), ..., d(n-n_d))\end{aligned} \tag{2.2}$$

where $\hat{y}$ is the response time, f is the non-linear function, u is the number of requested containers, z is the number of running containers, d is the number of users and n is the discrete time index.

Neural Networks

Neural Networks are employed extensively for system identification due to their satisfactory performance in non-linear dynamic system modelling [52]. In this work, we use the Multi-Layer Perceptron (MLP) as a feed-forward neural network for system identifications. Figure 2.2 shows the proposed architecture of the MLP neural network. For simplicity, only one hidden layer is considered.

The first layer is the input layer, the second layer is the hidden layer, and the last one is the output layer. We use hyperbolic tangent as the activation function for the hidden layer, and the output layer is a linear function.

According to Figure 2.2, the input vector for the Neural Network is:

$$\begin{aligned}x_i(n) =& (u(n), ..., u(n-n_u), \\ & z(n), ..., z(n-n_z),\, d(n), ..., d(n-n_d))\end{aligned} \tag{2.3}$$

Where in our case, u, z, d, and n are the number of requested containers, number of running containers, number of users, and discrete-time index, respectively. Also, n_u, n_z, and n_d denote the corresponding time delay for each of these inputs.

The cost function for training the neural network model, intended for system identification purposes, is delineated as follows:

$$J = (y(n+1) - \hat{y}(n+1))^2 = \frac{1}{2}(e_m(n+1))^2 \tag{2.4}$$

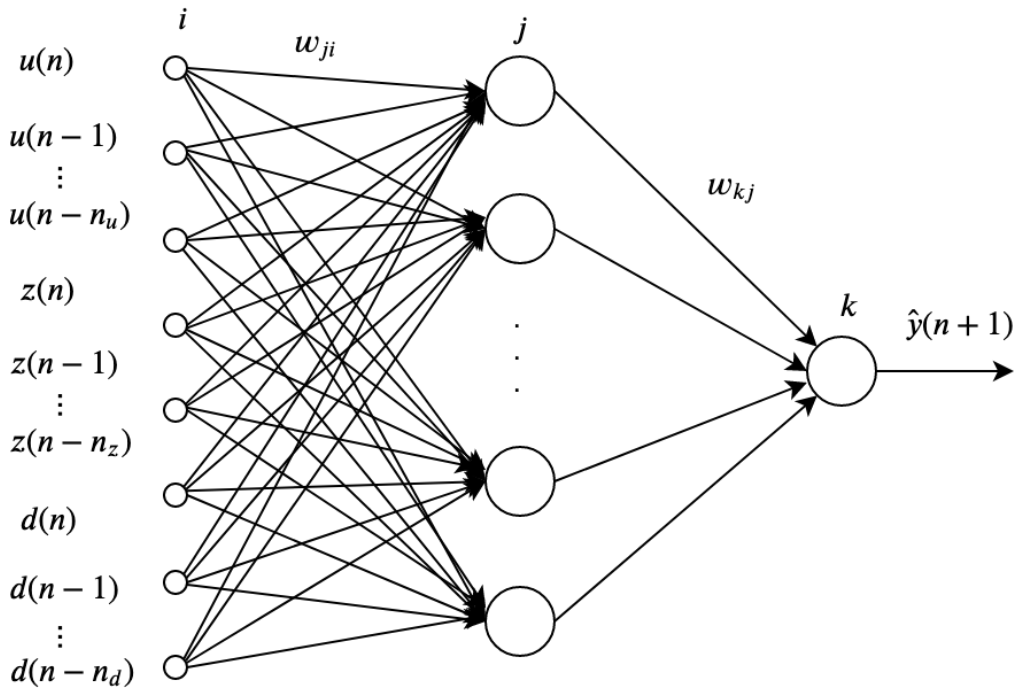


Figure 2.2: The structure of the proposed neural network for modelling the behaviour of the cloud software system.

where $y(n+1)$ represents the measured output, $\hat{y}(n+1)$ is the predicted output, and $e(n+1)$ is the prediction error.

The system identification procedure is required to obtain the sensitivity of the system's output to the change in control input, which is the number of requested containers in our case. The Jacobian of the system can be derived as follows:

$$\frac{\partial y(n+1)}{\partial u(n)} \approx \frac{\partial \hat{y}(n+1)}{\partial u(n)} \tag{2.5}$$

Data-driven system identification comprises two stages: 1) data collection and 2) model training and verification. In the following subsection, we discuss the best practices to collect the appropriate data for better system identification.

Data Collection

Data collection is an indispensable part of data-driven system identification, and the resultant performance of the modelling highly depends on the quality of the acquired

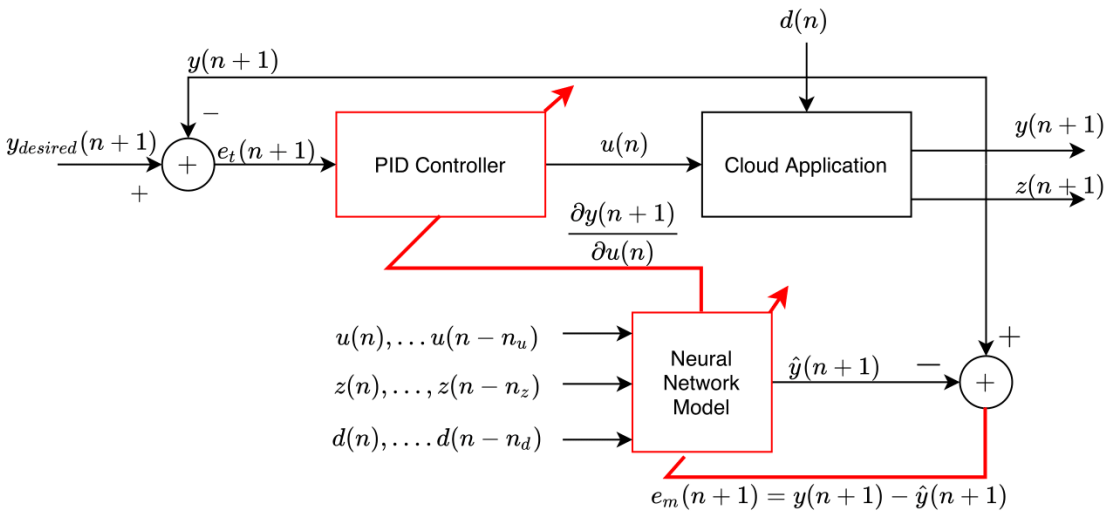
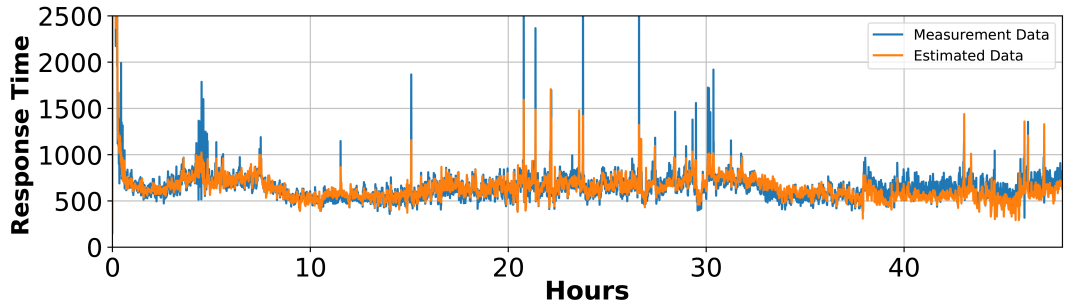


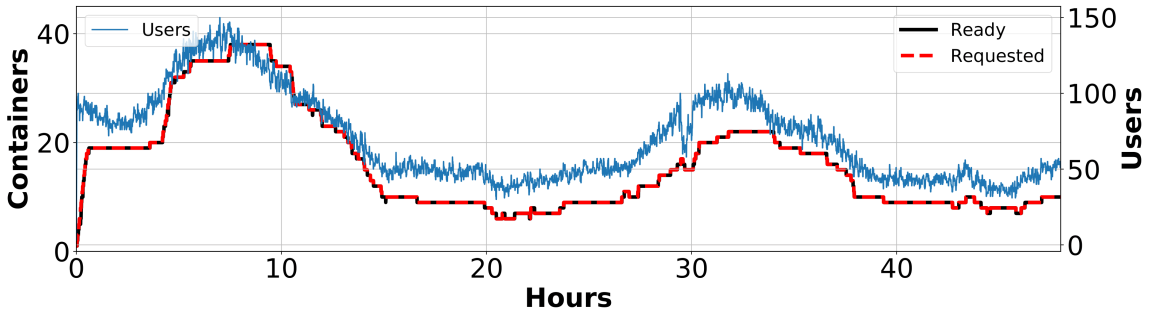
Figure 2.3: The overall structure of the identification and control of a cloud application, red arrows on the Neural Network model and PID controller blocks indicate the online adaptation of these blocks.

data. System identification can be carried out in an offline/online manner. Most of the time, it is preferable to train the neural networks model online. However, this requires a high sampling rate. Since the monitoring systems for cloud software systems suffer from a low sampling rate, we are bound to first use some offline data for pre-training, and then do further online training at run-time to adapt the model to the changes in application or underlying infrastructure. Hence, we have a model that can be used in a highly dynamic environment. To this end, we have collected offline data from a cloud software system as our system under study to be identified for about 48 hours and with a sample rate of 20 seconds. To acquire meaningful data from the system, it is crucial to monitor the system's behaviour in most system states. Furthermore, we can apply the data logged in the cloud monitoring system's repository for identification purposes. One can find an example of data logging module in our online repository[1].

[1]https://github.com/pacslab/NNPIDAutoscaler



(a) Measurement vs estimated value from the trained model.



(b) The dynamics of workload and the number of running containers during the off-line training.

Figure 2.4: Training the neural network before activating the adaptive controller. During this time, the system uses a pure reactive auto-scaler to bring the adaptive controller up to speed.

Model Training

In this section, we use data collected in the data collection phase (refer to Section 2.5.1) to obtain the performance model. A multi-layer perceptron with one hidden layer containing 40 neurons is adopted for this purpose. According to Equation (2.3), the input vector consists of two-time delays for u, z, and d, i.e., $n_u = n_z = n_d = 2$. Therefore, the input vector is a nine-by-one vector. The number of time delays is obtained from observations in data. The number of neurons, hidden layers and time delays depends on the non-linearity of the target software system. The more non-linear the system, the more neurons and deeper architecture are needed. Therefore, we selected the number of neurons and hidden layers after several manual training and finding the best fit. These numbers depend on the non-linearity of the target software system. We tried to use the simplest model since increasing the number of neurons,

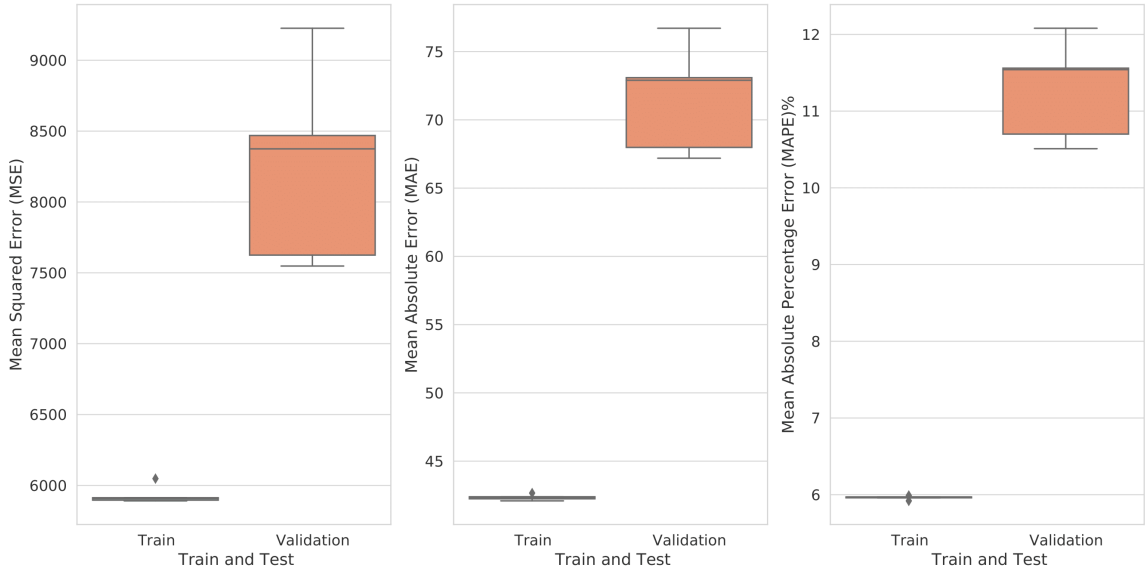


Figure 2.5: Distribution of the evaluation metrics for performance modelling in the training and validation stages.

and hidden layers may result in the problem of overfitting and computational overhead. Our goal is to model the non-linear behaviour of the cloud software system with these nine inputs for the next time step.

In order to train the model, 67% of the dataset is used for training and 33% for the validation. In the pre-processing step, we scaled the input data to the range 0 to 1 according to the Min-Max scaling algorithm, to prevent the neural network model from saturating. We used Adam Optimizer to train the weights and biases of the neural network with a training rate of $\eta = 0.002$, $\beta_1 = 0.99$, $\beta_2 = 0.9$, and $\epsilon = 1e - 8$ which are the default training values for the optimizer [121]. We set the batch size to 32 and the training epoch to 10000. We executed the model training five times.

Now, we can respond to **RQ2.1**. Figure 2.5 shows the distribution of the evaluation metrics for performance modelling in the training and validation stages. In our offline training phase, our neural network showed a Mean Squared Error (MSE) of 5931.20 and 8248.60, a Mean Absolute Error (MAE) of 42.34 and 71.57, and a Mean Absolute Percentage Error (MAPE) of 5.96% and 11.278% for the training set and the validation set, respectively. This is while the average response time in our training set is 667.79

ms, with a variance of 187,480.34.

The root cause of the reported modelling errors can be due to a lack of sufficient and representative data or an appropriate regularization in the model.

Considering the variations in the data and the obtained evaluation metrics from the model, our neural network provides acceptable predictions during the training and evaluation phases for an extremely stochastic system. The training set and validation set results show that the neural network is not over-fitting to the training set during our training.

2.5.2 Controller Design

In this study, we opted to use the adaptive PID controller's incremental version, which is available at our online repository [2]. In this scheme, the error signal for the controller is defined as follows and is shown in Figure 2.3:

$$e_t(n+1) = y_{des}(n+1) - y(n+1) \tag{2.6}$$

where $y_{des}(n+1)$ is the desired set point for the response time of the cloud software system, and $y(n+1)$ is the measured response time of the system.
Control law for the discrete incremental PID [86] is defined as follows, one should note that since number of containers cannot take float values we use ceiling function on the $u(n)$:

$$\begin{aligned} u(n) = \lceil u(n-1) + k_p(e_t(n) - e(n-1)) + \\ k_I e(n) + k_D(e(n) - 2e(n-1) + e(n-2)) \rceil \end{aligned} \tag{2.7}$$

where $u(n)$ is a control input at discrete-time n. k_p, k_I, and k_D are the proportional, integral, and derivative gains of the incremental discrete controller respectively. We are going to adaptively change these gains as the system behaviour changes with respect to the following cost function for controller [86]:

$$J = \frac{1}{2}(y_{des}(n+1) - y(n+1))^2 = \frac{1}{2}(e_t(n+1))^2 \tag{2.8}$$

[2] https://github.com/pacslab/NNPIDAutoscaler

To adjust the controller gains, the gradient descent method has been applied as follows:

$$\begin{aligned} k_P(n+1) &= k_P(n) - \eta_c \frac{\partial J}{\partial k_P} \\ k_I(n+1) &= k_I(n) - \eta_c \frac{\partial J}{\partial k_I} \\ k_D(n+1) &= k_D(n) - \eta_c \frac{\partial J}{\partial k_D} \end{aligned} \tag{2.9}$$

where η_c is the controller learning rate. From equations 2.8 and 2.9, using the chain rule for derivatives, we can derive the following equations for updating the PID controller gains:

$$\begin{aligned} \frac{\partial J}{\partial k_P} &= \frac{\partial J}{\partial y(n+1)} \frac{\partial y(n+1)}{\partial u(n)} \frac{\partial u(n)}{\partial k_P} = -e_t(n+1) \frac{\partial y(n+1)}{\partial u(n)} \theta_1(n) \\ \frac{\partial J}{\partial k_I} &= \frac{\partial J}{\partial y(n+1)} \frac{\partial y(n+1)}{\partial u(n)} \frac{\partial u(n)}{\partial k_I} = -e_t(n+1) \frac{\partial y(n+1)}{\partial u(n)} \theta_2(n) \\ \frac{\partial J}{\partial k_D} &= \frac{\partial J}{\partial y(n+1)} \frac{\partial y(n+1)}{\partial u(n)} \frac{\partial u(n)}{\partial k_D} = -e_t(n+1) \frac{\partial y(n+1)}{\partial u(n)} \theta_3(n) \end{aligned} \tag{2.10}$$

where $\frac{\partial y(n+1)}{\partial u(n)}$ is Jacobian of the controlled system and it is obtained from the system identification process, and $\theta_1(n) = e(n) - e(n-1)$, $\theta_2(n) = e(n)$ and $\theta_3(n) = e(n) - 2e(n-1) + e(n-2)$. The algorithm for proposed adaptive controller is shown in Algorithm 1.

It is worth noting that the controller adaptation will be performed when the neural network exhibits "satisfactory accuracy". In our experimental evaluations, we define the neural network is satisfactory accurate when the prediction error in response time is less than 300ms. Considering this value helps us prevent controller adaptation from diverging since if any large changes happen in the identified model, it will lead to a wrong Jacobian. According to Algorithm 1, the NN model is fine tuned at runtime to adapt itself to the changes in the environment, therefore, we can wait until the model is learned online again and then adjust the controller to the new model. Moreover, according to Algorithm 1, the controller optimization is only performed when the neural network model's prediction error is lower than the threshold and

when the regulation error is not zero, i.e. the response time is not between 500-800 ms. Although a full stability analysis would stray from the scope of this study, the proposed approach for adaptation of the controller is a viable and robust means to prevent the same adaptation from driving the closed-loop system toward instability.

2.6 Evaluation

In this section, we describe the experimental setup and the configuration of the underlying infrastructure. Then, we discuss the experimental methodology for more detail on workloads and performance metrics.

2.6.1 Experimental Setup

To properly evaluate the effect of the proposed algorithm on performance compared to with other auto-scaling algorithms, we deployed a three-tier containerized WordPress cloud application with MySQL as the database and Nginx as the webserver deployed on a Kubernetes cluster. We picked WordPress as the benchmarking application due to its widespread use in more than 34% of all websites over the Internet [225]. For load testing, we leveraged an extended version of Locust Library [149], available publicly in our repository[3] on the same cluster to minimize the effect of network latency in the results. Our extended version of Locust gave us more flexibility during experiments than any other available load testing tool. We used a simple REST API developed on the load testing tool[4] that helped us control the load testing and measure the performance of the application at runtime. The details of our deployment, along with the configuration settings, can be found in our GitHub repository[5]. To synchronize the WordPress's upload folder between different instances, we used an NFS server deployed to the same zone as the Kubernetes cluster to minimize the network latency.

[3]https://github.com/pacslab/pacs_locust
[4]https://github.com/pacslab/pacs_load_tester
[5]https://github.com/pacslab/wordpress-kubernetes-deployment

Algorithm 1: Proposed Adaptive Controller Algorithm

input : y_{n+1} — The average response time measured for the current time step.

input : $u_n, u_{n-1}, ..., u_{n-n_u}$ — The number of requested containers for n_u previous time steps.

input : $z_n, z_{n-1}, ..., z_{n-n_z}$ — The number of running containers for n_z previous time steps.

input : $d_n, d_{n-1}, ..., d_{n-n_d}$ — The number of users for n_d previous time steps.

input : $K_P(n), K_I(n), K_D(n)$ — The controller parameters at time step n.

output : $u(n+1)$ — The updated controller command for the next time step.

while *True* **do**

`/* According to the current parameters(`$K_P(n), K_I(n), K_D(n)$`) */`

Calculate $u(n)$ using Equation 2.7

Calculate the Jacobian using Equation 2.5

Prepare the input vector $x_i(n)$ for NN model according to Equation 2.3

`/* NN model predicts the average response time. */`

$\hat{y}(n+1) \leftarrow f(x_i(n))$ according to Equation 2.2

$e(n+1) \leftarrow \hat{y}(n+1) - y(n+1)$

`/* Checking the accuracy of the predictor. */`

if $|e(n+1)| < Threshold$ **then**

`/* Updating the control variables */`

$k_P(n+1) \leftarrow k_P(n) - \eta_c \frac{\partial J}{\partial k_P}$

$k_I(n+1) \leftarrow k_I(n) - \eta_c \frac{\partial J}{\partial k_I}$

$k_D(n+1) \leftarrow k_D(n) - \eta_c \frac{\partial J}{\partial k_D}$

else

`/* The control variables are not updated due to unsatisfactory prediction. */`

$k_P(n+1) \leftarrow k_P(n)$

$k_I(n+1) \leftarrow k_I(n)$

$k_D(n+1) \leftarrow k_D(n)$

end

`/* According to the new control parameters(`$K_P(n+1), K_I(n+1), K_D(n+1)$`) */`

Calculate $u(n+1)$ using Equation 2.7

`/* To improve the prediction accuracy the NN-Model is fine-trained. */`

Fine-tune the NN-Model by optimizing Equation 2.4

end

Table 2.1: Configuration of the Kubernetes Cluster.

Property	**Value**
Cluster Zone	us-central1-a
Minimum Nodes	1
Maximum Nodes	7
VM Type	n1-standard-1
vCPU	1
RAM	3.75 GB
SSD	30GB
OS	Container-Optimized OS
Client Version	1.13.9
Server Version	1.13.6

Kubernetes Cluster

For the purpose of this work, we deployed a Kubernetes cluster on the Google Cloud Platform (GCP) using the Google Kubernetes Engine (GKE). The cluster configuration used in the experiments can be found in Table 2.1.

Benchmarking Application

In this section, we introduce the details of our benchmarking three-tier containerized application. For this study, we used WordPress with PHP FPM version 7.3 as our application server. Our application server uses MySQL version 5.6 as the database and Nginx version 1.7.9 as its web server. Our configuration files, as well as the Docker images and deployment procedure, is publicly available on the WordPress deployment GitHub repository mentioned above.

2.6.2 Experimental Methodology

In this section, we discuss different workloads and the motivations behind selecting them. Moreover, the performance metrics of interest are presented to compare the controllers' ability to satisfy the cloud software system SLA requirements. It is noteworthy that before designing the experiments, we first evaluated the available resources' capacity in handling the number of users. Regarding the VMs, we could create 42 containers of the WordPress Application. Therefore, the number of containers is more than sufficient to satisfy the control objectives,e.g. SLA. We assumed that the orchestration system works correctly in resource provisioning, i.e. the infrastructure is reliable. In other words, all the containers are created successfully when the appropriate command is called.

Step-change Workload

Figure 2.6 portrays the step-change workload (blue line) in which the number of users changes abruptly, resembling a step signal. The number of users rises from 30 to 60 and then declines back to 30, in a period of 36 minutes. The rationale behind considering such workload is to evaluate the robustness of the controllers against sudden disturbances and also study the adaptive behaviour of the proposed controller in managing the SLA, performance in particular. This is a common situation for online websites that offer short time deals, attracting a considerable number of users in a short time.

FIFA World-Cup Workload

Figure 2.6 presents the second workload (black line). In this case, the number of users varies according to the variations of the FIFA World-Cup 1998 data set [16] collected from '1998-06-30 08:00:01' to '1998-07-01 08:00:00'. Given the systems' capacity in handling the number of requests, we scaled down the number of users between 30 and 150 while persevering the shape of the trace. The reason for choosing

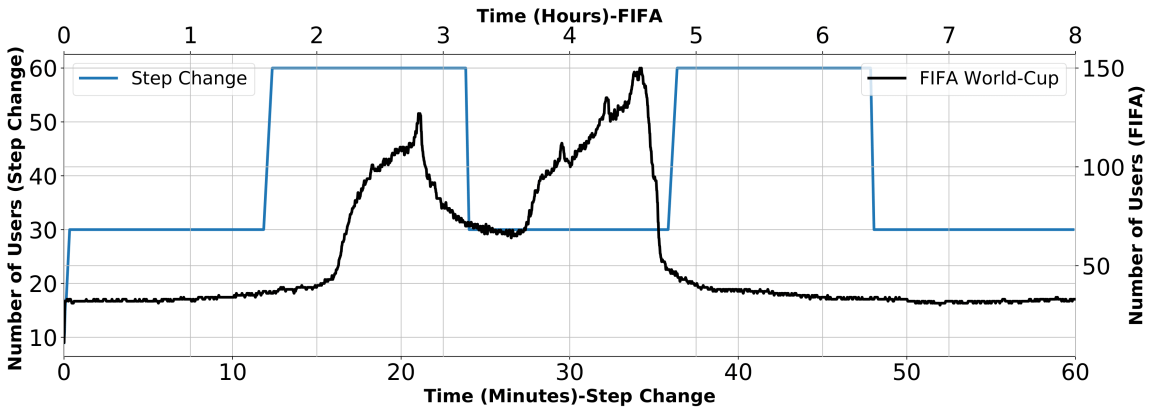


Figure 2.6: FIFA World-Cup and Step Change Workload.

150 as the number of user upper limit is that the application is close to its saturation point but can still maintain the control target in the desired region. Also, the time scale is transformed from 24 hours to 8 hours, which helped us study the control system's performance under more intensified fluctuations in the number of users. This behaviour may occur in web services offering live sports game streaming. At the start of each sport event, the number of online streamers increases gradually, and at the end of the event, a gradual decrease can be observed.

Performance Metrics

Various performance metrics are considered to have a clear comparison of controllers' capability and efficiency in maintaining the optimized performance. We consider well-known performance metrics from the control and software community to have the best of two worlds. The Mean Squared Error (MSE), Mean Absolute Error (MAE), and rise time are used as representative performance indices from control theory. The definition of MSE and MAE are given in Equation (2.11) and (2.12), respectively. For the sake of simplicity, we define the rise time as the time required for the controller to bring the response time of the system to the desired operating region after the disturbance has been applied.

$$MSE = \frac{1}{N}\sum[e(t)]^2 \tag{2.11}$$

$$MAE = \frac{1}{N}\sum|e(t)| \tag{2.12}$$

For software domain performance metrics, Service-Level Agreement (SLA) and efficient provisioning are considered. We define the SLA as follows: the upper limit for the average response time ($1,000ms$) cannot be violated more than one minute, i.e., three consecutive monitoring intervals. If such a violation occurs, we aggregate the violation time and count this as a violation with a penalty. Since the SLA does not change frequently, i.e. the set point for the response time is not changed; we are dealing with a regulation and disturbance rejection control problem. Therefore, the goal is to regulate the set-point around the defined SLA and reject the effect of change in the number of users on the response time,i.e. disturbance rejection.

Three different provisioning efficiency metrics are considered according to the Cumulative Distribution Function (CDF). The over-provisioning is defined as the percentage of the time that response time is below 500ms, i.e., containers are more than necessary. We prefer this value to be as low as possible. The under-provisioning is defined as the percentage of the time that response time lies between $800 - 1,000ms$, i.e., more containers are required—similarly, the less the under-provisioning, the better. Efficient provisioning is defined as the percentage of the time that response time is between 500-800ms, i.e., the number of containers is according to the system requirements. It is ideal to have higher values for efficient provisioning.

2.6.3 Experimental Results

To evaluate the efficiency of the proposed method in maintaining the response time of the three-tier containerized web application, several experiments have been carried out with different controllers; in particular, the responsiveness and robustness of the controllers under two types of workloads are investigated.

The scaling heat algorithm is employed from [25] as the baseline, which represents a robust reactive algorithm to maintain the performance metrics. According to this algorithm, when a violation of the upper threshold occurs, indicating the saturation of the container, we increase the heat factor by one, and if a violation from the lower threshold happens, indicating resource under-utilization, we decrease the heat factor by one. If no violations occur, i.e. the utilization is within the range, we decrease the heat factor by one until the heat factor is zero. If the heat factor is equal to a specific number (e.g., $n/-n$), we create/terminate one container. The primary distinction between the scaling heat algorithm and other threshold-based algorithms is that it waits for several consecutive violations to react. This gives the scaling algorithm robustness against the ping-pong effect. The ping-pong effect is an undesirable alternating scale up and scale down of the resources (containers), which results in oscillation in the number of containers and the performance metrics. In our experiments, we chose the number of consecutive violations needed for the scaling to occur to be 5 ($n = 5$), according to the [25], with an upper trigger point of 800 and a lower trigger point of 500.

As for the non-adaptive control theoretical approach, we evaluate the fixed-PI (FPI) and fixed PID (FPID) controllers. Note that in the absence of an appropriate linear model for the target software system, these controllers' gains are obtained after a short period of experimentation and manual tuning. The gains were set to $K_P = 0.0004$, $K_I = 0.0004$, and $K_D = 0.00005$.

The obtained gains from the manual training are used as an initial condition for our proposed adaptive controllers. We consider adaptive neural network based PI (NN-PI) and adaptive neural network based PID (NN-PID) controllers for evaluation. The controller learning rate η_c is set to $2e-13$ for K_P and K_I, and $2e-14$ for K_D, which is tuned manually. The reason for setting small values for these control parameters and the learning rate is because the time is in millisecond and choosing larger values may result in divergence in adaptation. Adam Optimizer is selected for training the

weights and biases of the neural network with the training rate of $\eta = 0.002$, $\beta_1 = 0.99$, $\beta_2 = 0.9$, and $\epsilon = 1e - 8$. There is no specific way to find the optimal initial value for controller learning rates, and we need to find it by trial and error.

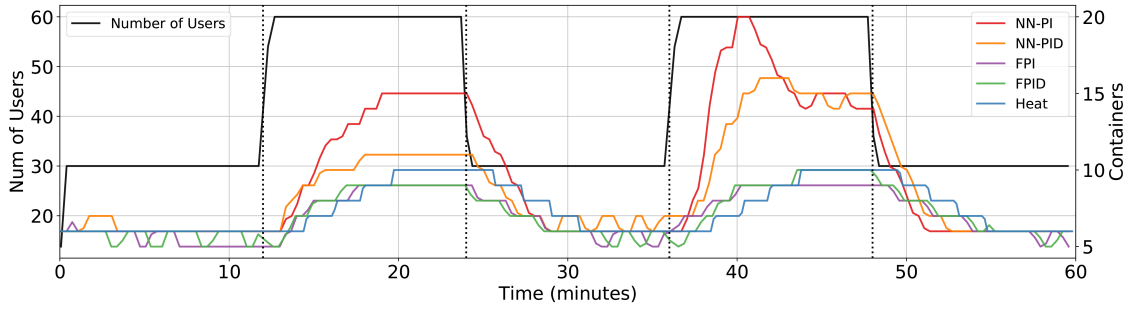
The control objective is to maintain the response time of the system between 500-800ms (set-point) by adjusting the number of containers (actuator) to immediately react to the change in the number of users (disturbance). In this range, the control error is considered zero, and hereafter we refer to this range as the desired operating region. The motivation behind choosing this range is that most of the cloud providers prefer to maintain the response time of their system in a pre-defined range, to prevent SLA violations and over-provisioning at the same time. Therefore, we can evaluate the controllers' applicability to the real world control problems for cloud service providers.

For each workload, we conduct the experiment three times for reproducibility. All the reports are given on average of these three experiments. Care has been taken to carry out the experiment in similar conditions. In other words, we conduct the experiments each day simultaneously and on the same infrastructure. In our experiments, we used Google N1 standard machine types. These types of virtual machines do not use shared CPUs, i.e. each of them has its own dedicated resources. Therefore there is minimal interference between Virtual Machines, which is negligible. According to GCP benchmarks, these instances only have 2.47% standard deviation regarding their performance [82]. Finally, we compare and discuss the performance metrics for all these controllers and enumerate the pros and cons of each.

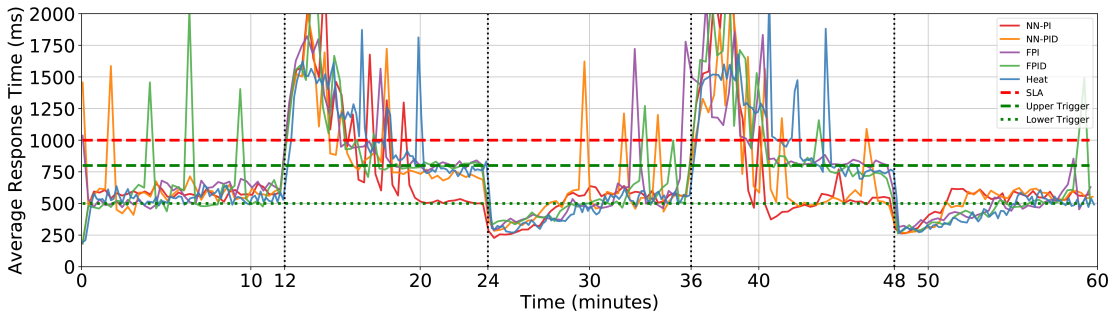
2.6.4 Step-Change Workload

This experiment's main objective is to investigate the efficiency of the controllers against abrupt disturbance imposed on the system. Besides, we aim to observe the effect of the PI and PID controllers' adaptability when facing a recurrent disturbance in the future.

Figure 2.7b depicts the response time of the system under study for the first



(a) Variations in the number of containers imposed by the variation in the workload.

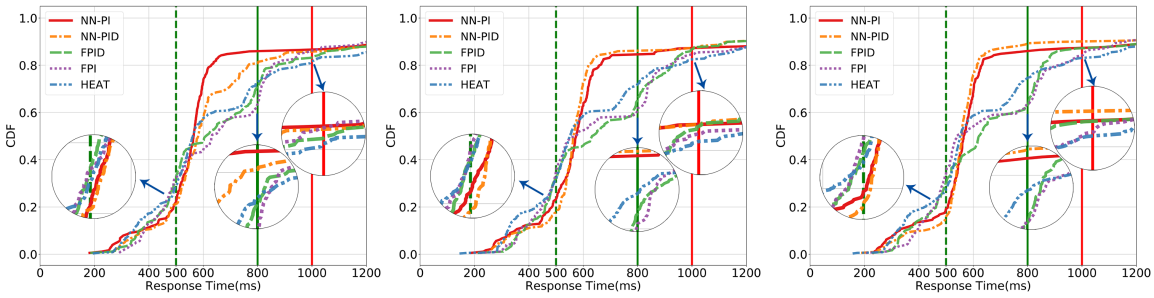


(b) Response time of the cloud software system for step change in number of users. Vertical dashed lines indicate the time of these step changes.

Figure 2.7: Comparison of five controllers in managing the response time of the cloud software system.

experiment. Upper and lower trigger points are shown by horizontal dashed and dotted green lines, respectively. The dashed red line shows the SLA limit for the response time.

Figure 2.7a shows the control signal (i.e., the number of successfully provisioned containers) generated by each controller in response to the disturbance (i.e., number of users) injected to the system. In Figure 2.8, Cumulative Distribution Function (CDF) is shown separately for each conducted experiment. In this figure, the dashed/solid vertical green line shows the lower/upper trigger point of the desired operating region. Also, SLA is shown by a solid red line. Figure 2.9 shows the changes in the adaptive controller parameters in response to the step-change workload. According to this figure, the controller parameters' changes become smaller with respect to time, indicating the convergence to an optimal value. Moreover, since the controller parameters are



(a) First experiment (b) Second experiment (c) Third experiment

Figure 2.8: Cumulative Distribution Function (CDF) for all the experiments.

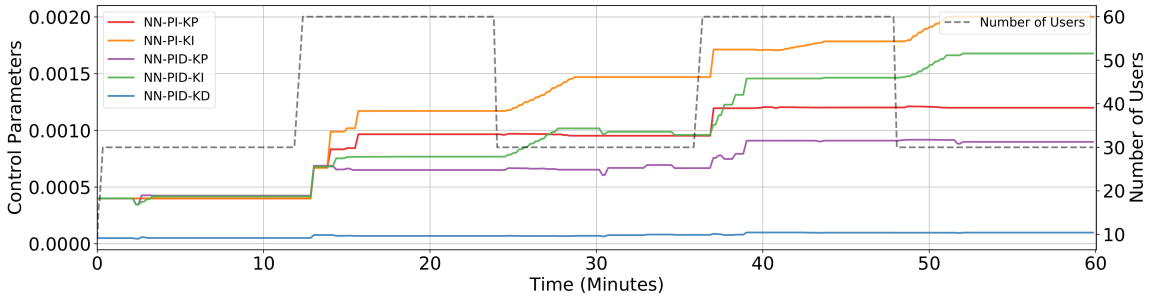


Figure 2.9: Adaptation in control parameters for Step-Change workload.

adjusted when the number of users changes, we can conclude that the NN model has a good prediction ability.

Table 2.2 shows a summary of the results by presenting the average and the Standard Deviation (STD) of the three experiments performed in this study. Small STD for these three experiments indicates that the results are reproducible. Table 2.3 demonstrates the improvements achieved by adaptive controllers, compared to the scaling heat algorithm.

Response Time: According to Table 2.2, all controllers have average response-time laying in the desired operating region.

Consequently, we need to do a deeper investigation on other metrics.

Provisioning Efficiency: According to Table 2.2, NN-PID and NN-PI significantly decreased the resource over-provisioning ratio.

According to Table 2.3, our results indicate that adaptive controllers achieve an improvement of 34.46% and 22.28% for handling under-provisioning for NN-PID and

NN-PI, respectively. This finding suggests that adopting the proposed method uses less resources and, consequently, less energy. This fact can be seen in Figure 2.8. Magnification around lower trigger point (dashed green line) reveals that both NN-PI and NN-PID are more capable of handling the problem of over-provisioning.

As reported in Table 2.3, similar behaviour is observed in the efficient provisioning of the resources. NN-PID and NN-PI were successful in assigning an efficient amount of resources to the cloud software application. According to the efficient provisioning percentage reported in Table 2.3, the efficiency of the resource allocation improved by 74.31% for NN-PID and 65.83% for NN-PI compared to the scaling heat algorithm. This observation can be made looking to both magnified lower and upper trigger points in Figure 2.8.

Similarly, one can confirm these improvements by looking at the system's behaviour around the upper trigger point and SLA, which is magnified for better perception in Figure 2.8.

From Table 2.2, leveraging adaptive controllers, we observed an increase in the average number of containers used over the experiments. This is mainly due to the fact that efficient resource provisioning doesn't necessarily lead to using fewer containers or to less changes in the resources used. On the contrary, the control algorithm should instantly provide enough resources according to the control objective.

Service Level Agreement: This comparison is mainly based on the description of SLA in Section 2.6. According to Table 2.2, results confirm that control-theoretical controllers are more capable of preventing SLA violations. And among them, adaptive controllers comply with the SLA requirements more effectively. This fact is highlighted when comparing the improvements in SLA violations. As can be seen, compared to the scaling heat algorithm, SLA violations decreased by 43.2% and 49.26% for NN-PI and NN-PID, respectively.

Rise Time: In Section 2.6.2, we described the rise-time and the reason behind considering this performance metric. In this experiment, we have four rise-times,

i.e., two rising edge disturbance t_1 and t_3, and two falling edge disturbance t_2 and t_4. Based on the results presented in Table 2.2, control-theoretical approaches considerably outperform the scaling heat algorithm. Moreover, adaptive controllers exhibited better disturbance rejection characteristics compared to the fixed PI and fixed PID controllers.

Evaluating this performance metric revealed two interesting observations. Considering the scaling heat algorithm as the baseline in Table 2.3, the first observation is that there is a considerable improvement in the rise time of the system adopting the proposed adaptive controller. For example, we observed an almost 50% improvement in the control system's rise time for the first rising edge disturbance. The second observation is that the second rising/falling edge disturbance's effect becomes less severe, confirming the controller's optimized behaviour for similar disturbances. This shows the adaptability of the proposed controller when dealing with future disturbances. This phenomenon can be observed in Figure 2.7a. After the first rising edge disturbance, NN-PI and NN-PID have the highest slope in creating containers. This slope gets even steeper for the second rising edge disturbance. Similar behaviour is seen for the falling edge.

Table 2.2: The average and standard deviation of the performance metrics for three conducted experiments for step-change workload.

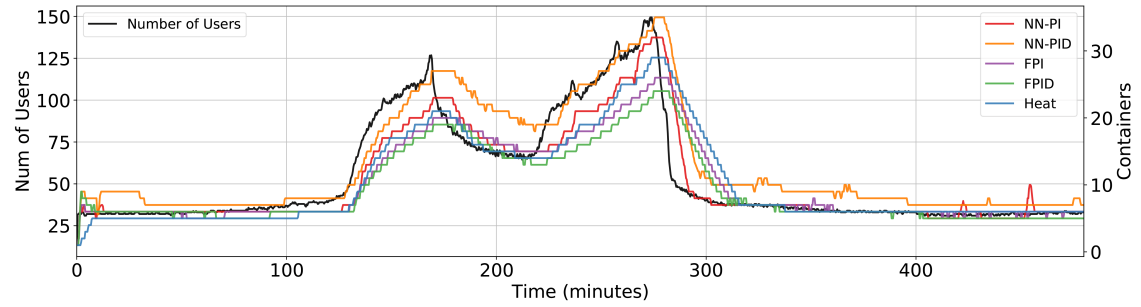
	Controller Type				
Performance Metrics	Heat	F-PI	NN-PI	F-PID	NN-PID
avg. RT[STD] (ms)	699.82[11.30]	729.52[5.16]	672.98[10.48]	734.97[6.01]	667.07[27.19]
avg. Containers[STD]	7.50[0.02]	7.11[0.06]	9.26[0.30]	7.06[0.10]	8.86[0.06]
Over-Provisioning[STD] %	28.00[2.70]	33.88[1.18]	21.76[2.35]	31.27[0.87]	18.35[2.82]
Efficient Provisioning[STD] %	38.46[6.06]	28.83[2.27]	63.78[2.84]	35.20[4.06]	67.04[7.07]
Under-Provisioning[STD] %	9.20[1.07]	22.28[1.41]	1.31[0.85]	10.09[6.46]	2.24[2.45]
SLA Violations[STD] (s)	540.00[20.00]	413.33[11.55]	306.67[64.29]	366.67[23.09]	340.00[87.18]
t_1[STD] (s)	460.00[34.64]	386.60[50.33]	240.00[40.00]	373.40[90.18]	233.40[30.55]
t_2[STD] (s)	386.60[11.54]	293.40[11.55]	260.00[0.00]	346.60[46.18]	233.40[41.63]
t_3[STD] (s)	413.34[41.63]	373.40[100.66]	140.00[20.00]	340.00[138.56]	146.60[[11.55]
t_4[STD] (s)	393.40[11.54]	306.60[41.63]	160.00[20.00]	300.00[40.00]	140.00[40.00]

Table 2.3: Improvement in the performance metrics for the first experiment compared to the scaling heat algorithm, all reported in percentage.

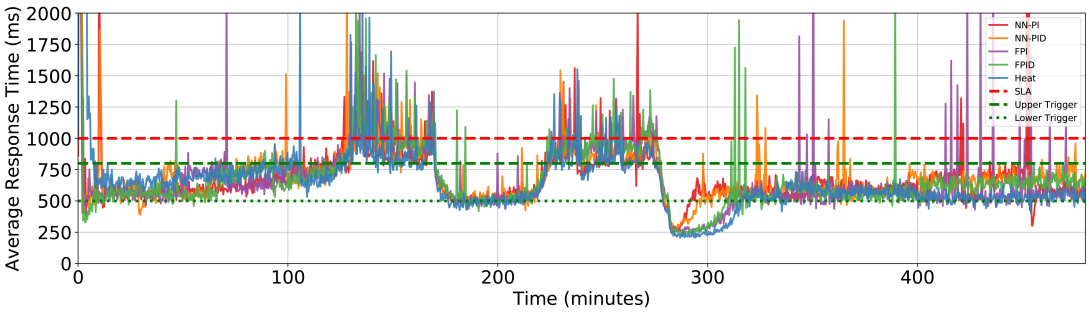
	Improvement %	
Performance Metrics	NN-PI	NN-PID
Efficient Provisioning %	65.83	74.31
Under-Provisioning %	85.76	75.65
SLA Violations %	43.20	37.03
t_1 %	47.82	49.26
t_2 %	32.74	39.62
t_3 %	66.12	64.53
t_4 %	59.32	64.41

Discussion: Taking into account all the performance metrics, we can respond to research questions 2 and 3. For **RQ2.2**, all controllers could maintain the average response time in the desired operating region. However, the scaling heat algorithm could not adequately react to the sharp disturbances, accounting for more SLA violations. On the contrary, control theoretical methods had fewer SLA violations and reacted to the disturbances faster. Amongst control-theoretical approaches, the proposed adaptive controllers significantly outperformed the non-adaptive counterparts in both SLA and rise time.

Regarding **RQ2.3**, considering the discussion made in provisioning efficiency, it can be concluded that adaptive controllers are more capable of handling resource management tasks by drastically improving the over-provisioning, Efficient provisioning, and under-provisioning. Furthermore, we can see that both controllers show satisfactory performance compared to the adaptive PI and PID performance. This experiment motivated us to evaluate them against a more realistic workload.



(a) Variations in the number of containers imposed by the variation in the workload.



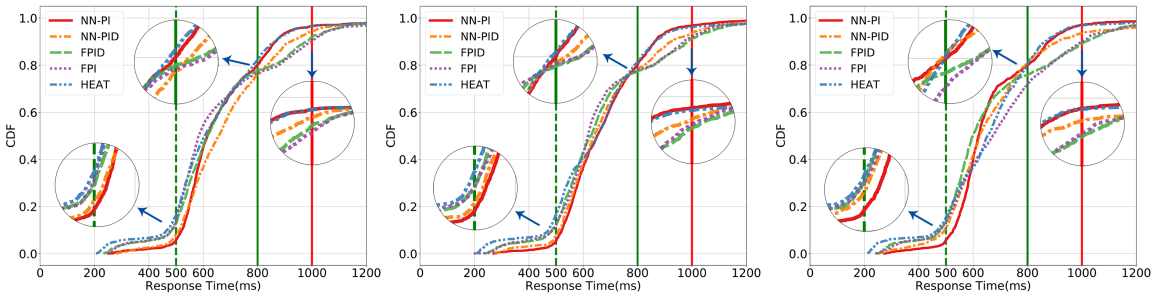
(b) Response time of the cloud software system for FIFA Wold-Cup workload.

Figure 2.10: Comparison of five controllers in managing the response time of the cloud software system against real world workload.

2.6.5 FIFA World-Cup Workload

In this experiment, exposing the cloud software system to the second workload described in Section 2.6.2, we aim to examine the performance of the proposed controllers against a more realistic workload for a longer time interval. The average response time is shown in Figure 2.10b. All of the thresholds are defined similarly to the step-change experiment. Figure 2.10a depicts the changes in the number of containers in response to the workload variations.

Additionally, Figure 2.11 presents the cumulative distribution function of response time for all these three experiments conducted for reproducibility validation. Figure 2.12 shows the changes in the adaptive controller parameters during the experiment. According to this figure, the variations in the controller parameters decrease with respect to time. This behaviour shows that the controller parameters are being



(a) First experiment. (b) Second experiment. (c) Third experiment.

Figure 2.11: Cumulative Distribution Function (CDF) for all the experiments.

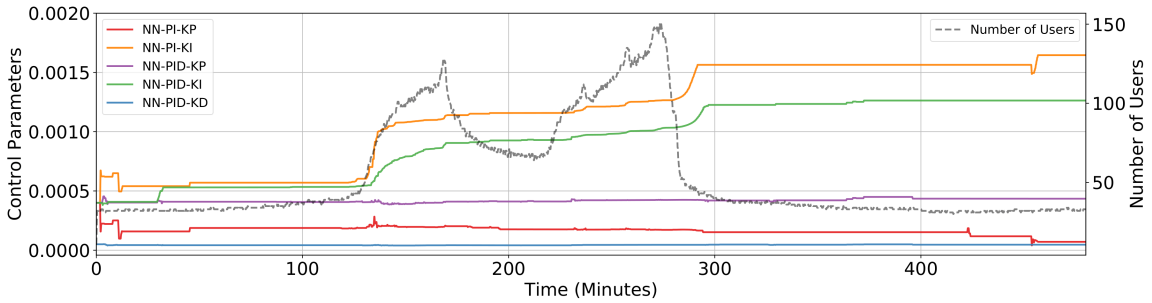


Figure 2.12: Adaptation in control parameters for FIFA World-Cup workload.

optimized, and the proposed adaptive controller is working properly. Details of the experiments are summarized in Table 2.4 along with the STD of three experiments. Small STD for these three experiments indicates that the results are consistent. Furthermore, assuming the scaling heat algorithm as the baseline, improvements in the metrics are reported in Table 2.5 for comparison.

Response Time: According to Table 2.4 and Table 2.5, the results are in accordance with the findings of the previous experiment on the step-change workload. All the controllers could maintain the average response time in the desired operating region.

Provisioning Efficiency: Referring to Table 2.4, significant improvement is observed for the proposed adaptive methods in terms of the resource over-provisioning. Control theoretical approaches tend to allocate resources more cost-effectively. The over-provisioning drops from 16.04% to 5.66% and 8.87% for NN-PI and NN-PID, respectively, which results in 64.71% and 44.7% cost reduction. For the fixed versions

of PI/PID controllers, this cost reduction is around 15%, which still is significant at a large scale. This can be observed in Figure 2.11 around the lower trigger point shown by the dashed green line. The magnified picture is presented for better perception. NN-PID and NN-PI showed lower over-provisioning. On the contrary, the scaling heat algorithm suffers from high resource over-provisioning.

In terms of efficient provisioning, results substantiate the capabilities of the adaptive controllers in efficient resource management. According to Table 2.4, all control-theoretical methods allocated the resources more efficiently compared to the scaling heat algorithm. Results in Table 2.5 suggest 22.12% and 14.21% better resource allocation for NN-PI and NN-PID, respectively. Referring to Figure 2.11, around the upper trigger point (solid dashed line), all controllers maintain the response time below the upper trigger point for almost 75% of the time. A similar performance is observed in all experiments.

Although the performance was not ideal for under-provisioning, results reveal that NN-PI and NN-PID are conservative in allocating more resources than the scaling heat algorithm. This fact can be seen in Table 2.5 that shows the increase in under-provisioning from 15.19% to 16.76% and 15.28%. However, this increase is negligible, and results show that it does not affect the SLA. Therefore, we can infer that adaptive controllers tend to use the resources close to the system capacity, without violating the SLA.

Service Level Agreement: As can be seen in Table 2.4, surprisingly, the scaling heat algorithm shows a decent performance compared to Fixed PI/PID and NN-PID in terms of the number of SLA violations. However, adopting NN-PI decreased the SLA violations by 27.11%, compared to the scaling heat algorithm. We found that since in control-theoretical approaches, "controller gains" decide about the magnitude of change in the number of containers, non-adaptive controllers are not suitable for slowly varying workloads. In the absence of a system model, these gains cannot be found accurately. And even if we have a model, we need to adapt the model to

the environment changes. This fact highlights the need for leveraging an adaptive controller.

Table 2.4 suggests an inferior performance for the adaptive PID controller. This result could be because the auto-scaler has to deal with the cloud software system's stochastic behaviour, the presence of disturbances, noisy measurements, and delay in the actuator. Having a derivative term would increase the controller's sensitivity to these artifacts, leading to system instability.

According to Table 2.4, all controllers tend to use the same number of containers on average except NN-PID, which used 23.13% more containers. Besides, according to Figure 2.10a, this controller applies a greater change in the number of containers compared to other controllers with the expense of degraded performance (e.g., SLA). The reason for this is that creating/removing containers will add a computational overhead to the system. This influences the response time of the system for a while, and if this happens a lot, it will drastically deteriorate the overall performance. This behaviour contributes to the reason why 80% of the controllers prefer not to use the derivative term in the controller [11].

Mean Squared Error: Mean squared error plays a crucial role in designing the adaptive neural network-based PI/PID controller. The adaptive behaviour of the controllers is achieved by minimizing the MSE. According to Table 2.5 and Table 2.4, the optimization task is carried out successfully by causing 56.27% and 35.81% drop in MSE compared to the scaling heat algorithm, for NN-PI and NN-PID, respectively. This improvement can be observed in Fixed PI/PID as well. The reason is that the control-theoretical approaches' main objective is to decrease the system error with respect to its magnitude. On the contrary, the scaling heat algorithm ignores the magnitude of the error.

Mean Absolute Error: In addition to MSE, Mean Absolute Error (MAE) was reduced by 42.72% and 14.10% for NN-PI and NN-PID, respectively.

Note that MSE denotes the controller's performance in regulating the response

time of the system and shows whether these regulation errors are significant or not in magnitude, having more emphasis on larger errors. As opposed to MSE, MAE looks at the errors with similar weights and ignores the magnitude.

Table 2.4: The average and standard deviation of the performance metrics for three conducted experiments for FIFA workload.

	Controller Type				
Performance Metrics	Heat	F-PI	NN-PI	F-PID	NN-PID
avg. RT[STD] (ms)	674.28[11.21]	693.54[19.13]	680.76[6.26]	684.62[1.05]	698.14[5.01]
avg. Containers[STD]	10.46[0.058]	11.33[1.61]	10.51[0.37]	10.35[1.12]	12.88[0.82]
MSE[STD] ($\times 10^4$)	11.12[1.30]	6.40[0.40]	4.86[0.48]	6.20[1.08]	7.14[1.10]
MAE[STD]	68.79[1.65]	76.06[3.16]	39.40[2.36]	72.89[4.86]	59.09[7.10]
Over-Provisioning[STD] %	16.04[0.50]	13.29[0.94]	5.66[0.20]	12.73[0.81]	8.87[1.84]
Efficient Provisioning[STD] %	60.94[0.63]	62.29[3.87]	74.42[0.25]	64.25[1.76]	69.60[1.31]
Under-Provisioning[STD] %	15.19[0.96]	14.99[2.70]	16.76[0.37]	13.75[0.53]	15.28[2.84]
SLA Violations[STD] (s)	393.33[57.74]	1,860.00[336.45]	286.67[50.33]	1,873.33[323.31]	873.33[133.16]

Table 2.5: Improvements in the second experiment, compared to the scaling heat algorithm.

	Improvement %	
Performance Metrics	NN-PI	NN-PID
avg. Response Time %	-0.96	-3.53
avg. Containers %	-0.47	-23.13
Mean Squared Error %	56.27	35.81
Mean Absolute Error %	42.72	14.1
Over-Provisioning %	64.71	44.70
Efficient Provisioning %	22.12	14.21
Under-Provisioning %	-10.33	-0.59
SLA Violations %	27.11	-122.03

Discussion: Exposing the controllers to a realistic workload for a longer time revealed interesting findings. For **RQ2.2**, the same behaviour in maintaining the average response time was observed, confirming the results of the previous experiment.

Also, better MSE and MAE were achieved by the control-theoretical approaches. However, surprisingly, fixed PI/PID exhibited unacceptable performance in terms of the SLA violations. In contrast, the adaptive PI controller significantly decreased the SLA violations compared to the other controllers.

Regarding the provisioning efficiency (**RQ2.3**), the control-theoretical approaches present better resource management capabilities in almost all cases. However, the scaling heat algorithm performs better in terms of under-provisioning. An interesting observation is the tendency of the adaptive PID in creating/removing the containers, which causes performance overhead and, consequently, SLA violations.

Comparing both adaptive controllers, we can conclude that *adaptive PI* performs better than *adaptive PID* for real-world workloads and long term control purposes. This is mainly due to the fact that it is less sensitive to the uncertainties and noise in the system. Besides, it provides better resource management utilizing an almost equal number of containers while achieving fewer SLA violations.

2.7 Limitations

There are some limitations on this work. First, since we require an initial condition for our proposed controller, finding better initial conditions can lead to better performance. However, in the long-run, the optimization task will push the system to a minimized cost function. Furthermore, considering online model identification and the highly stochastic behaviour of the cloud software systems, noisy measurements could result in incorrect online system identification or controller divergence. It is better to select the identification and controller training rate as low as possible to address this issue. However, model training and controller optimization will take a longer time.

The second limitation is the extent of the generalizability of our experiments. In this study, we selected the WordPress application as a standard three-tier application for evaluating the adaptive behaviour of our proposed adaptive PID-controller in optimizing the performance of the containerized cloud software systems. The rationale

behind this decision is that WordPress is one of the main drivers for many websites nowadays. It worth noting that the focus of this study is to propose an adaptive control theoretical approach for optimizing the auto-scalers at runtime and the way we can benefit from a massive amount of collected performance data to obtain a non-linear model of the cloud software system, which can be used to optimize the performance and the auto-scaler. To the best of our knowledge, our study is the first to adopt neural networks to propose a model suitable for control theoretical auto-scalers, and we aim to pave the road for future studies in the area of cloud software system performance optimization. We expect the same efficiency and the same outcome for other applications. However, future studies may further investigate the impact of selecting different applications on the neural networks' architecture.

The third limitation of this study is the sampling rate selection. The sampling rate plays a crucial role, both in the system identification and the control mechanism. Increasing the sampling rate helps improve the reaction time to any change in the measurements while adding some overhead to the system monitoring and data acquisition. Moreover, a high sampling rate will not necessarily improve the controller's performance due to the presence of uncertainty in the measurements. Selecting different sampling rates results in different system models; therefore, we cannot make a fair comparison between these models. As a result, we relied on a widely used sampling rate, a sampling rate of 20 seconds, in the cloud infrastructures. Also, considering that the proposed controller itself is more complicated than a simple heat-based method, we impose a computation overhead to the system, which is not measured here.

Another limitation of this study is the stability analysis for the proposed controller. According to [206], the stability of a software system guarantees the ability of the system to converge to the objectives. A system can be stable without goals; however, a goal cannot be achieved in an unstable system. In our study, we defined this goal as the controller's ability to maintain the system's response time in a specific range, in other words minimizing the regulation error. For different software qualities,

we have different interpretations of stability. Also, to analyze a system's stability mathematically, we require the dynamic model of the system. Since our model is data-driven, we cannot directly use it to find the attraction region of the controller.

2.8 Conclusion and Future Work

In this research study, we proposed and evaluated various adaptive and non-adaptive controllers for containerized cloud applications. Through extensive experimentation, we identified the best adaptive controller that can optimize the performance and SLA under different parameter settings and configurations. To this end, we leveraged the power of black-box modelling, i.e., neural networks and PID controllers, to make controllers adaptive, scalable, efficient, and extendable to other containerized cloud applications. The key is to use a simple, pure reactive auto-scaler at the beginning of the software system operation and then hand over the auto-scaling to the adaptive controller when it has reached satisfactory accuracy in prediction. Afterward, the software system's performance under control will be kept optimized due to the neural network's online training and modifying the controller's parameters accordingly at run time.

Our experimental results show that the proposed adaptive PID controller works well even without a precise model for updating the controller's parameters. This makes them well suited for controlling cloud software systems.

For future studies, we plan to investigate the performance of different data-driven modelling approaches, such as Long Short Term Memory (LSTM), and introduce other optimization terms in our proposed adaptive controller's cost function,e.g. constraints on the number of containers. Moreover, we aim to study the effect of different sources of uncertainty on the performance of self-adaptive systems such as actuator uncertainties.

Chapter 3

Studying the Performance Risks of Upgrading Docker Hub Images: A Case Study of WordPress

3.1 Abstract

The Docker Hub repository contains Docker images of applications, which allow users to do in-place upgrades to benefit from the latest released features and security patches. However, prior work showed that upgrading a Docker image not only changes the main application, but can also change many dependencies. In a performance-critical production environment, one generally avoids changing multiple dependencies at once, since it may give rise to hard-to-diagnose performance degradation issues. In this chapter, we present a methodology to study the performance impact of upgrading the Docker Hub image of an application, thereby focusing on changes to dependencies. We demonstrate our methodology through a case study of 90 official images of the WordPress application. We observe that the performance of WordPress is mostly mandated by the dependency versions that are used by its official image, rather than the main application code itself. We further demonstrate how our methodology can be used to investigate the impact of major, minor and patch upgrades of WordPress' two main dependencies (Apache and PHP) on its performance. Our study shows that Docker image users should be cautious and conduct a performance test before

upgrading to a newer Docker image in most cases. Our methodology can assist them to better understand the performance risks of such upgrades, and helps them to decide how thorough such a performance test should be.

3.2 Introduction

The last decade has seen enormous advances in cloud computing, such as faster development cycles, better security and a lower cost of utilizing cloud-based resources. One core-enabling technology for cloud computing is the containerization of applications. Most modern cloud-based applications are dependent on many services and applications. This dependency gives rise to several issues, e.g., a conflict between the dependencies, missing dependencies, and platform differences [161]. A containerization technology, such as Docker, addresses these problems by packaging software code along with its dependencies to run on any computing environment or infrastructure [93]. A containerized application runs in fully isolated environments called *containers*.

The growth in the use of containerized applications has captivated researchers' attention to several aspects of this technology. Most research in this field studies Docker containers and repositories from the security [42, 57, 207, 246] and storage management [213, 251] perspective. For instance, Shu et al. [207] show that both official and community images on Docker Hub have security issues, and that the vulnerabilities propagate from parent images to the child images. Such security issues can often be addressed by upgrading the Docker image. However, upgrading a Docker image may change many dependencies at once. In a study of 37K Docker images, Gholami et al. [80] showed that a median of 8.6 dependencies change in one image upgrade. As prior work showed that changes to dependencies can cause quality issues for the software that depends on them [108, 163, 186], there is always the challenge of deciding how a Docker image upgrade will affect the quality of the containerized application.

One quality aspect that is increasingly important for modern software is perfor-

mance – Harman and Hearn even state that it should now be considered "the new correctness" [87]. However, performance testing is not a widely implemented practice in industry [33], or by developers [135] in general. Our hypothesis is that the same applies to Docker images. While in cases, the individual performance of the software components inside the image may have been tested, their performance most likely has never been tested when they operate in tandem.

In this research study, we propose a methodology for better understanding the performance risks of upgrading a Docker image, thereby focusing on its dependencies. We demonstrate our methodology through a case study of a performance-critical application, WordPress. First, we conduct load tests on 90 images from the official WordPress Docker Hub repository. These 90 images span a total of 27 WordPress versions with different combinations of dependencies (i.e., ranging from PHP 5.6.x to PHP 7.x.x and from Apache 2.4.10 to 2.4.38). Second, we demonstrate how our methodology can be used to investigate the changes in the performance of the WordPress application between these images. We demonstrate our methodology through the following research questions (RQs):

- **RQ3.1:** What is the impact of upgrading the Docker image of WordPress on its performance?

  Motivation: We analyze how the performance of WordPress changes through several types of upgrades of its official Docker image. This analysis shows which image upgrades are problematic in terms of performance.

  Results: There are considerable variations in average response time for images with the same WordPress version, implying that the changes in its dependencies cause performance variations. We observed that doing a patch to patch upgrade in the Docker images of WordPress results in unpredictable impact, causing up to a 340% drop in performance or an improvement of up to 77%. The same behaviour was observed for minor and major upgrades as well.

- **RQ3.2:** What is the relation between the performance of the WordPress application and updates of its dependencies?

  Motivation: As we could not attribute the performance changes to a specific WordPress version or type of upgrade in RQ3.1, we investigated how changes in the main dependencies of WordPress (Apache and PHP) were correlated with the changes in performance.

  Results: We observed that all studied WordPress Docker images that use PHP with major version 5 have a considerably lower performance. Hence, upgrading an image with PHP version 5.6.x to one with PHP 7.0.x improved the performance for all studied images. Also, upgrading to the next available PHP patch version degraded WordPress' performance in 71 out of 123 (58%) of the cases. We observed that upgrading the last patch version in a minor/major version to the first available patch version in a minor/major version of PHP (e.g., from 7.1.33 to 7.2.01) always improved the performance of WordPress. Finally, we observed that doing a minor to minor upgrade was more likely to result in performance improvement than a patch to patch upgrade. We did not observe a relation between changes to the Apache dependency and the performance of WordPress.

Our case study shows how our methodology can be employed to study the performance risks of a Docker image upgrade. For example, our case study shows that WordPress users who are planning to upgrade their WordPress image with PHP version 5.x.x to 7.0, can do so safely from a performance point of view.

The remainder of this chapter is organized as follows. Section 3.3 discusses the background concepts of the research study and outlines the related work to the study. In addition, we give background information about our case study subject (WordPress). Section 3.4 presents our methodology for analyzing the performance risks when upgrading a Docker image. Section 3.5 presents the results of our case study. Section 3.6 provides suggestions and recommendations for practitioners and

researchers. In Section 3.7, we discuss the threats to the validity of our work, and we give directions for future research based on those threats. Section 3.8 concludes the chapter.

3.3 Background and Related Work

In this section, we introduce the essential technologies used throughout the study and discuss the prior work that is related to our study of the performance risks of upgrading Docker Hub images. Finally, we give background information about our case study subject (WordPress).

Docker

As an open-source container system, Docker provides a lightweight, low overhead, fast and efficient solution to implement containerized applications [161]. Before Docker, installing, and deploying software on different environments was a painstaking task. Leveraging Docker, one can pull the application images from a repository such as Docker Hub, or one can build their own image and deploy it. Docker images consist of several layers. The first layer is the base image, which is the foundation of the application. Additional layers can be added on top of the base image. The instructions to create the image are specified in the Dockerfile, in which each instruction corresponds to a layer in the image. This Dockerfile can be used by others to recreate that Docker image from a base image. A Docker container is a runnable instance of a Docker image [35], which consists of all the required dependencies required for an application to perform correctly.

Docker Hub

Docker Hub [62] is a repository for sharing Docker images. In May 2021, Docker Hub hosted more than seven million Docker images. These images are distributed using private or public repositories, which provide a convenient way for software

versioning. Public repositories are divided into groups, namely, official and community. Official repositories provide authentic releases of the Docker image of an application since they are reviewed and published by a team that is sponsored by Docker Inc. A Docker image can be retrieved from a repository by its tag. For example, `wordpress:5.2.2-php7.1-apache` is a Docker image for WordPress version 5.2.2 with PHP version 7.1 and the latest supported version of Apache. Many applications on Docker Hub provide convenience tags for their images, such as `wordpress:latest`, which always points to the latest WordPress image.

Docker Performance Analysis

With the increased use of Docker containers in the cloud, performance evaluation of these containers is getting more attention. Felter et al. [69] extensively studied the performance of a native Linux environment, Docker and KVM. They evaluated these environments in the presence of CPU intensive, I/O intensive, and network-intensive workloads. They concluded that Linux containers provide better or equal performance in contrast to VMs in almost all cases. Lingayat et al. [146] evaluated the performance of deploying Docker containers on bare-metal and virtual machines and provided a clear sketch of the necessity of deploying Docker containers in bare-metal environments. Lingayat et al. reported about 50% performance gain when running the containerized Docker applications on bare-metal as opposed to virtual machines. They attributed the performance degradation of deploying the application on VMs to the architecture of VMs, specifically since VMs run on emulated hardware, introducing additional layers and consequently unnecessary performance overhead. Ruan et al. [194] analyzed the performance of containers in the cloud and presented several recommendations for developers to determine which container to use for different usage scenarios. Ruan et al. conducted various experiments to measure the difference in performance among application containers (e.g., Docker) and system containers (e.g., LXC). Furthermore, they assessed the overhead of an additional layer of the VM between the bare-metal

and containers. They pointed out that system containers can better manage I/O extensive workloads than application containers, and that running containers in the VM environment would result in a degradation of disk I/O performance up to 42.7%, and of network latency up to 233%. Casalicchio et al. [46] investigated the available tools for measuring Docker's performance from the host operating system and virtualization environment perspective. They presented a characterization of the CPU and disk I/O overhead introduced by containers. After evaluating four open-source performance profilers (`mpstat`, `iostat`, `docker stats`, and `cAdvisor`) they pointed out that characterizing the overhead and workload is not a straightforward task due to the instability and lack of dedicated tools for measuring a wide range of performance metrics. Their results show that available container monitoring tools give different results and should be properly interpreted. Xu et al. [242] evaluated the performance of deep learning tools in Docker containers. Many deep learning software frameworks have been developed and updated frequently to benefit from new hardware features and software libraries. They evaluated Docker containers' impact on the performance of deep learning applications by benchmarking I/O, CPU, and GPU in Docker containers. Xu et al. pointed out that the Docker engine minimizes the introduced overhead by extra layers added between applications and hardware resources. They concluded that encapsulating deep learning tools into Docker containers can address the problems mentioned earlier and performs as well as the host system. However, it is more flexible, lightweight, and provides resource isolation. While there are several studies on the performance of Docker, we are the first to study the performance impact of upgrading Docker images.

Performance Evolution

A closely-related study area is performance evolution, in which the performance of a system is studied as it evolves. In the remainder of this section, we give a non-exhaustive overview of prior work on performance evolution.

Part of prior work on performance evolution focused on how to conduct load tests for multiple software versions. For example, Alcocer [7] proposed a multi-dimensional profiler Rizel as a way to test different software versions by adjusting variables of the execution context such as function inputs and benchmarks. They showed in a case study that their proposed method can successfully identify the piece of code that introduced a loss of performance. Additionally, Alcocer et al. [8] conducted an extensive performance analysis to study the evolution of the performance of 49 benchmarks along with the evolution of 1,439 versions from a variety of 19 open-source projects. They identified the patterns for performance regression and performance improvement by contrasting differences of source code and variation in the execution patch and calling context tree. Mostafa et al. [168] studied a framework for "Performance-aware" repository and revision control for Java programs. They introduce the PARCS system that automatically tracks the behavioural differences across revisions of a program to figure out the impact of changes on its performance.

In addition, a large body of prior work on performance evolution focused on identifying performance regressions through repository analysis techniques in various types of software systems [4, 10, 31, 32, 65, 73, 92, 152, 171, 172].

Former studies have always focused on performance changes that are caused by the source code of the system itself (e.g., to identify performance regressions). In contrast, we focus on performance changes that are caused by the environment in which the system runs (e.g., by its dependencies).

3.4 Methodology

In this section, we present our methodology for analyzing the performance risks of upgrading Docker images. Our methodology consists of the following steps (depicted by Fig. 3.1): (1) collecting image information for WordPress from the Docker Hub image repository, (2) deploying WordPress and identifying the used dependency versions, and (3) collecting and analyzing the performance data. Each of the steps will be

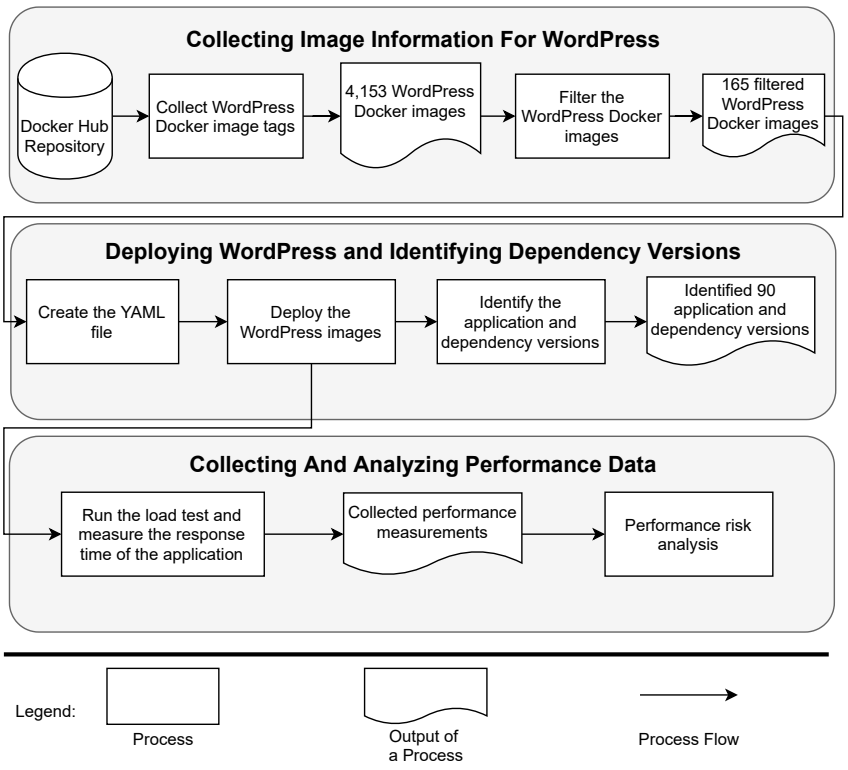


Figure 3.1: Overview of the empirical study design.

discussed in the following subsections.

3.4.1 Collecting Image Information for WordPress

In this section, we discuss the steps taken to collect the WordPress Docker images and the inclusion criteria for the Docker images.

Collect WordPress Docker Image Tags: As the first step, a web crawler was developed to automatically collect the WordPress Docker image tags from the official WordPress Docker Hub repository[1]. Since Docker images with a WordPress version before 4.7.2 did not have information regarding the operating system and architecture or were not working after deploying, we did not include them in our data set.The crawler collected 4,153 WordPress Docker image tags, including all different WordPress Docker images with different operating systems and architectures.

Filter the WordPress Docker Images: Since not all collected WordPress Docker

[1]https://hub.docker.com/_/wordpress

images are deployable on our available infrastructure, such as images for the `Arm` architecture, we defined several filtering criteria. The inclusion criteria for Docker images are as follows:

- The image must be an official Docker image.
- The image must target the `amd64` architecture.
- The image must depend on the Apache web server rather than Nginx.

After applying these filtering criteria, we selected 165 WordPress Docker images to conduct the performance test and identify the dependency versions.

3.4.2 Deploying WordPress and Identifying Dependency Versions

In this section, we elaborate on the developed two-tier containerized WordPress application and procedure to identify the application and dependency versions at runtime.

To carry out the performance analysis, we employed three Virtual Machines (VMs), namely Cluster Master and Kubernetes Node 1 and 2. The cluster configuration used in the experiment can be found in Table 3.1. All these VMs are deployed on the Cybera cloud[2] with the same resources and configuration. Fig. 3.2 presents the overview of the Kubernetes cluster. The cluster master manages the deployment of application and services in a cloud environment. The first Kubernetes node contains the WordPress application with the Apache webserver in one container, and MySQL in another container. The second Kubernetes node, contains the Locust [149] application for load testing and monitoring. The rationale behind using seperate VMs for Locust and the main application is that generating users adds overhead on the running VM, which may affect the performance of the application under study. The MySQL (version 5.6) and Locust versions are kept constant throughout the experiment. We

[2]https://www.cybera.ca/rapid-access-cloud/

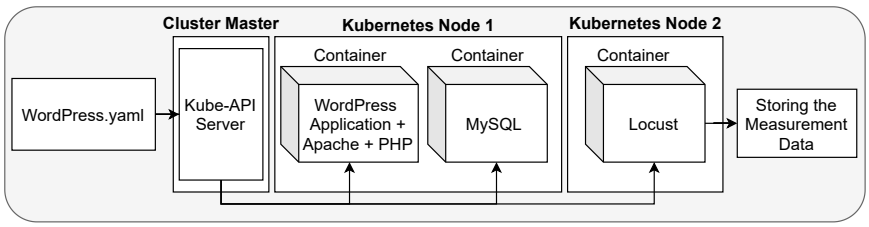


Figure 3.2: The deployment setup of the WordPress images.

Table 3.1: Configuration of the Kubernetes Cluster.

Property	**Value**
Cluster Location	Edmonton
VM Type	Medium
vCPU	2
RAM	3.75 GB
Storage	40 GB HDD
OS	Ubuntu
Client Version	1.16.3
Server Version	1.16.3

used MySQL version 5.6 because it is still one of the most used versions of MySQL[3]. The WordPress images are changed through a configuration setting in the YAML file.

All the VMs run in the same network to reduce the effect of transmission time. Therefore, measuring the response time of the system will roughly show the service time of the system. All the deployment steps are executed in an automatic fashion. Therefore, we do not need to do anything manually, which is a convenient way of carrying out performance analysis across many Docker images.

Create the YAML File: The image information collected in the previous section is used to create YAML configuration files for the automated deployment of the applications. This YAML configuration file is created based on the selected WordPress Docker images in Section 3.4.1. Also, in this file, we allocate a 200 milli-core vCPU and 256MB of RAM for each WordPress container.

[3]https://www.eversql.com/mysql-8-adoption-usage-rate/

Deploy The WordPress Images: The created YAML file is sent to the cluster master to control the Kubernetes orchestration platform to first pull the Docker image from Docker Hub and then deploy and run the WordPress application.

Identify the Application and Dependency Versions: To identify the version used by the WordPress images, we used the "Wappalyzer" library [231]. This library can detect the version of the content management systems, e-commerce platforms, web frameworks, server software, and analytic tools at runtime. From all 165 images, 6 images could not be analyzed at runtime and were removed from the data set. After identifying the application and dependency versions of the WordPress Docker images, we realized that some of the Docker images used convenience tags for names and were using the identical dependency versions. For example, we identified 8 WordPress Docker images with WordPress version 5.3, Apache version 2.4.38 and PHP version 7.3.12, all with different image names. Therefore, we removed 69 duplicate images from our analysis.

Table 3.2 shows the 7 different minor versions of WordPress including the number of patch versions for each minor version. Table 3.3 shows the number of different minor versions for PHP and the corresponding patch versions. Table 3.4 shows the distribution of the Apache patch versions. Note that the number of images in these tables are larger than the number of patches. This is because there can be several images for a patch version that have different dependency versions, e.g., WordPress version 4.7.x contains three images with patch version 4.7.2 and two images with patch version 4.7.5 (both with different PHP versions). The complete list of included unique WordPress Docker images with their corresponding dependency versions is available online [12].

Table 3.2: Distribution of the studied WordPress versions.

WordPress Version	#Images	#Patches
4.7.x	4	2
4.8.x	12	4
4.9.x	35	9
5.0.x	15	4
5.1.x	6	2
5.2.x	15	5
5.3.x	3	1
Total	90	27

Table 3.3: Distribution of the PHP versions in the data set.

PHP version	#Images	#Patches
5.6.x	17	9
7.0.x	18	12
7.1.x	26	19
7.2.x	20	16
7.3.x	9	8
Total	90	64

3.4.3 Collecting and Analyzing Performance Data

Run the Load Test And Measure the Response Time of the Application: To acquire the performance data, an extended version[4] of Locust [149] is used for load testing purposes. This extended version can collect the data with a fixed sample rate and is able to calculate the average response time of the application for these fixed time windows. Furthermore, to send requests to the application and measure the performance metrics, we used the REST API of the load testing tool. Our tool is publicly available online [12].

To measure the performance of each image, we introduced 20 users to the WordPress

[4]https://github.com/pacslab/pacs_locust

Table 3.4: Distribution of the Apache versions in the data set.

Apache Version	**#Images**
2.4.10	34
2.4.25	44
2.4.38	12
Total	90

application through the Locust user generator, and then measured the response time of the application every two seconds and for 5 minutes. We repeated this test 3 times for every image to reduce the impact of variability in the cloud on our measurements (see Section 3.7 for more details). After studying the measurements, we observed that the measurements across the 3 executions were relatively stable. Therefore, we used only the measurements from the first execution in our analysis. We refer to these values as the average response times ($\overline{\text{RT}}$) hereafter. The type of the workload is a simple `HTTP GET` request to the main page of the WordPress website. The motivation behind selecting such a simple workload is to show that the performance can vary considerably even with the simplest workload. The WordPress website is using "Twenty Seventeen" as the default theme.

Performance Risk Analysis: Generally, the version number of software follows the "MAJOR:MINOR:PATCH" semantic versioning principle [181] (e.g. for version "7.2.15" 7, 2, and 15 are the major, minor, and patch version of the application, respectively). According to the semantic versioning definition, major updates will make incompatible API changes, minor updates will add functionality in a backward-compatible manner, and patch updates make backward-compatible bug fixes. Semantic versioning does not officially impose restrictions on how the performance of an application can be affected. However, our expectation is that because of the magnitude of the changes in each type of version, we can use changes in version numbers as a (rough) proxy for changes in performance. Based on the semantic versioning principle, we group the

images as follows:

- *Patch groups*: Images with the same major, minor and patch version of an application. For example, we group all the Docker images of WordPress with major version 4, minor version 8, and patch version 1 into the same patch group (WordPress 4.8.1). This group contains 3 images with PHP versions 5.6.31, 7.0.23, and 7.1.9 and all with Apache version 2.4.10.

- *Minor groups*: Images with the same major and minor version of an application. For example, the WordPress 4.8.x minor group contains all 12 images with major version 4 and minor version 8 of WordPress.

- *Major groups*: Images with the same major version of an application.

We create patch, minor and major groups for WordPress itself, and the two main dependencies of WordPress on which we focus in this study (PHP and Apache). Note that we only have one minor (2.4.x) and major (2.x.x) group for Apache.

After conducting the load test on each WordPress Docker image, we calculate the average response time for that WordPress Docker image as explained above. To investigate the performance risks of upgrading to that Docker image, we calculate the relative response time by comparing the average response time of that Docker image ($\overline{\text{RT}}_{\text{ver}_n}$) to that of all images in the previous ($\overline{\text{RT}}_{\text{ver}_{n-1}}$) patch, minor, or major group, as shown in Equation 3.1.

$$\overline{\text{RT}}_{\text{RelativeImprovement}}(\%) : \frac{\overline{\text{RT}}_{\text{ver}_{n-1}} - \overline{\text{RT}}_{\text{ver}_n}}{\overline{\text{RT}}_{\text{ver}_{n-1}}} \tag{3.1}$$

If the relative improvement is negative, it means that the performance has degraded; and if it is positive, there was an improvement in the performance. For example, upgrading a WordPress Docker image from 4.9.1 to another image with WordPress 4.9.2 resulted in 75.5% relative improvement, which means that this patch to patch

upgrade improved the average response time of the former WordPress Docker image by 75.5%.

3.5 Case Study Results

In this section, we present the results of our case study in which we apply our methodology for analyzing the performance risks of upgrading Docker Hub images to the WordPress web application. For each research question, we discuss the motivation, approach, and results accordingly.

3.5.1 RQ3.1: What Is the Impact of Upgrading the Docker Image of WordPress on its Performance?

Motivation: Using Docker images allows users to upgrade an application easily. However, instead of a careful, managed upgrade, one may be dealing with an upgrade that changes many dependencies as well. Prior work [80] showed that a median of 8.6 dependencies change when a Docker image is upgraded. These changes may have an impact on the main application's performance. In this RQ, we study how the performance of WordPress is affected by upgrading its official image from Docker Hub.

Approach: We grouped the WordPress images in patch, minor and major groups as explained in Section 3.4. We then conducted the following analyses:

- *Patch Version Analysis:* In the patch version analysis, we first analyzed the distribution of the average response times for each patch version group. We then analyzed the distribution of the relative response time improvements as a consequence of upgrading from one patch group to the next available patch group. We compute this by taking the Cartesian product of the images in both groups and computing the relative performance improvement according to Equation 3.1 for each upgrade in that product. In this analysis, we included upgrades from the last patch version in a minor version to the first available

version in the next minor version (e.g., from WordPress 4.7.5 to 4.8), as this would be the next 'natural' upgrade for that version.

- *Minor Version Analysis:* For the minor version analysis, we studied the distribution of the average response times for each minor version group. We then analyzed the performance change after upgrading from all images in a specific minor group (e.g., 4.8.x) to all images in the next minor group (e.g., 4.9.x). Similar to the patch version analysis, in this analysis, we included upgrades from the last minor version in a major version to the next major version (e.g., from WordPress 4.9.x to 5.0.x).

- *Major Version Analysis:* In the major version analysis, we look into the distribution of the average response times for each major version group. We then study the effect of upgrading from all images in a specific major version group (e.g. 4.x.x) to all images in the next major version group (e.g. 5.x.x).

To compare the distributions of the patch to patch, minor to minor and major to major upgrades, we employ the Mann-Whitney U test [154] with an α-value of 0.05. The null hypothesis of this statistical test is that the two input distributions are equal. If the p-value of the test is smaller than 0.05, the null hypothesis is rejected and we conclude that the difference between the distributions is statistically significant. In addition, we quantify the difference using Cliff's Delta effect size [150]. We use the following thresholds for Cliff's Delta d [192]:

$$\text{Effect size} = \begin{cases} \textit{negligible(N)}, & \text{if } |d| \leq 0.147. \\ \textit{small(S)}, & \text{if } 0.147 < |d| \leq 0.33. \\ \textit{medium(M)}, & \text{if } 0.33 < |d| \leq 0.474. \\ \textit{large(L)}, & \text{if } 0.474 < |d| \leq 1. \end{cases}$$

Patch Version Analysis of WordPress

There are considerable variations in average response time for images with the same patch version of the WordPress application. Fig. 3.3 shows the distribution of the average response times of the studied WordPress Docker images for each WordPress patch group. Each data point corresponds to the average response time of a specific WordPress Docker image. Fig. 3.3 shows that there is a considerable variation in the average response times of different Docker images for the same WordPress patch version. As shown in Section 3.7, this variation was not caused by variability in our measurements due to the cloud environment in which our experiments were executed. The variation implies that the performance of a group of WordPress Docker images with the same patch version of WordPress is not mandated by WordPress itself, but dependent on other component(s). In the next RQ, we will look further into how dependencies are related to the main application's performance.

In 158 out of 302 (52%) patch to patch upgrades, WordPress' performance is degraded. The last box plot in Fig. 3.4 shows all 302 possible consecutive patch to patch upgrades for the WordPress application. Fig. 3.4 shows that more than half of the possible patch to patch upgrades of the WordPress Docker images resulted in performance degradation. This finding shows that we cannot make accurate predictions about the effect of doing an image upgrade on the performance of the application based on the WordPress version of the image alone.

At worst, a patch to patch upgrade of a WordPress image resulted in 9% to 340% degradation in WordPress' performance. This degradation can be small such as when upgrading from WordPress 5.2.1 with PHP version 7.3.6 and Apache 2.4.25 to WordPress 5.2.2 with PHP version 7.1.32 and Apache version 2.4.38 (with a performance degradation of 9%). The worst-case occurred when upgrading WordPress 4.9.8 with PHP 7.2.12 and Apache 2.4.25 to WordPress 5.0.0 with PHP version 5.6.39 and Apache 2.4.25 (with a performance degradation of 340%). Clearly,

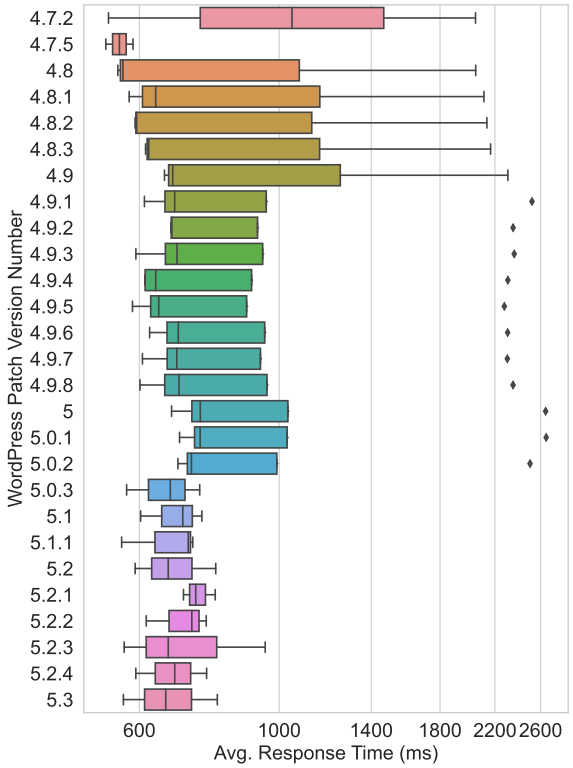


Figure 3.3: The distributions of the average response times for the studied patch versions of the WordPress application.

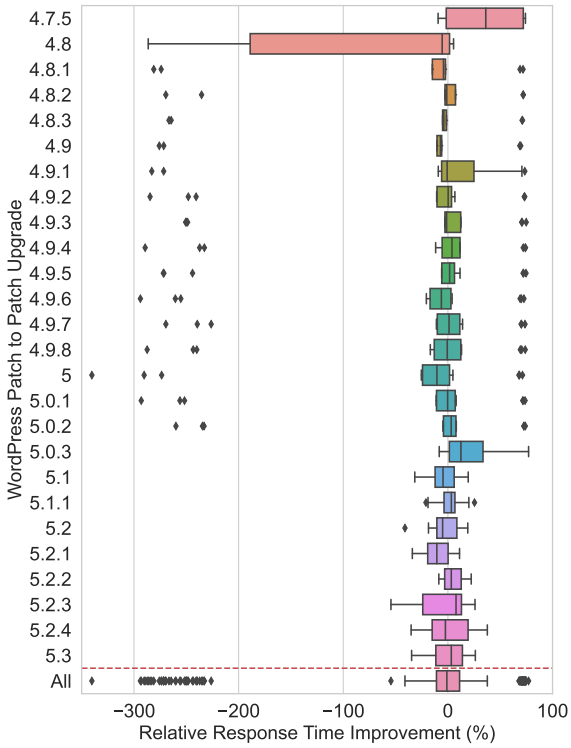


Figure 3.4: The distribution of relative response time improvements for patch to patch upgrades of the WordPress application images. The last box plot shows the distribution of relative response time improvements for all consecutive patch to patch upgrades of WordPress Docker images.

not all of these upgrades are intuitive ones (e.g., downgrading from PHP 7 to 5), but they do demonstrate the problems that can occur when deciding to upgrade solely based on the WordPress version in the image.

At best, a patch to patch upgrade resulted in a 5% to 77% improvement in the performance of the WordPress application. This improvement can be small (5%), such as upgrading from WordPress 4.7.5 with PHP version 7.1.5 and Apache 2.4.10 to WordPress 4.8 with PHP version 7.0.21 and Apache version 2.4.10. The improvement could also be large (77%), such as when upgrading from WordPress 5.0.2 with PHP version 5.6.39 and Apache 2.4.25 to WordPress 5.0.3 with PHP version 7.3.2 and Apache version 2.4.25.

The performance degradation was relatively larger when upgrading older patch versions of WordPress. Fig. 3.4 depicts the distribution of the relative

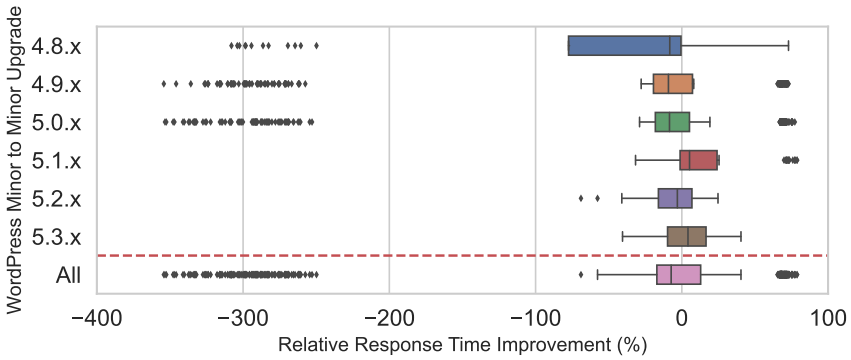


Figure 3.5: The distribution of relative response time improvements for minor to minor upgrades of the WordPress application images. The last box plot shows the distribution of relative response time improvements for all consecutive minor to minor upgrades of WordPress Docker images.

average response time improvements grouped by patch version. For example, in Fig. 3.4, 4.9.7 shows the distribution of relative average response time improvements after upgrading each WordPress Docker image in patch version 4.9.6 to each WordPress Docker image in patch version 4.9.7. Fig. 3.4 shows that doing a patch to patch upgrade before 5.0.2 may result in drastic performance degradation (e.g., up to 340% when upgrading version 4.9.8 to 5). However, the risk of getting a severe performance degradation became lower after upgrading to version 5.0.3. For patch versions released after 5.0.3 in the worst-case scenario, we observed a performance degradation of 54% when upgrading from a WordPress Docker image in 5.2.2 to a WordPress Docker image in 5.2.3. Nevertheless, for many applications – in particular performance-critical ones – 54% degradation is too large.

Minor Version Analysis of WordPress

In 786 out of 1,218 (65%) minor to minor upgrades of WordPress, the average response time is degraded. The last box plot in Fig. 3.5 shows the relative average response time improvements for all possible consecutive upgrades of the WordPress Docker image from a minor version to the next minor version. For example, we consider upgrading all WordPress Docker images with minor version 4.7.x

to 4.8.x and 4.8.x to 4.9.x and so on, resulting in 1,218 possible upgrades. Fig. 3.5 shows that 65% of these upgrades resulted in a lower performance than the previous version. Comparing this finding with its patch to patch counterpart, we observe that the risk of performance degradation is 13 percent points higher than a patch to patch upgrade, which was 52%. The Mann-Whitney U test shows that the performance risk of a minor to minor upgrade without conducting a performance test is significantly higher than for a patch to patch upgrade (albeit with a negligible effect size).

In the worst case, a minor to minor upgrade of the WordPress image resulted in 32% to 354% degradation of WordPress' performance. Fig. 3.5 presents the distribution of the relative response time improvements when upgrading the WordPress application from a minor version group to the next one. Fig. 3.5 shows that doing minor to minor upgrades may result in large variations in performance, for instance, upgrading to 4.8.x, 4.9.x, and 5.0.x from their previous minor version groups. This degradation may be small such as when upgrading from minor version 5.0.x to 5.1.x (with a performance degradation of 32%). The worst-case occurred when upgrading from minor version group 4.8.x to 4.9.x (which resulted in a 354% degradation of the performance).

In the best case, minor to minor upgrades improved WordPress' performance by 25% to 79%. Fig. 3.5 shows that there are some minor to minor upgrades that considerably improve the performance of the WordPress application. For example, there was a minor upgrade for the slowest WordPress Docker image in version 5.1.x to 5.2.x that improved the WordPress application's performance by 25%. Also, for a WordPress Docker image in version 5.0.x there was an upgrade to version 5.1.x, which resulted in a 79% boost in the performance. Similar to what the patch version analysis showed, it is hard to make predictions about the performance changes that are caused by an upgrade based on the WordPress minor version alone.

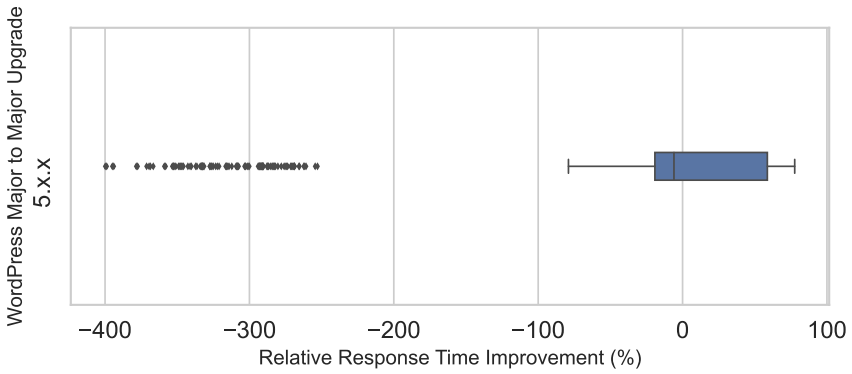


Figure 3.6: The distribution of the relative response time improvements for all consecutive major to major upgrades of WordPress Docker images.

Major Version Analysis of WordPress

Conducting a performance test is essential for WordPress Docker images even in a major upgrade. Fig. 3.6 shows the distribution of the relative response time improvements for all consecutive major to major upgrades of the WordPress application (from 4.x.x to 5.x.x, 1,989 cases).It is interesting to know that this upgrade can result in 400% performance degradation or 78% performance improvement. Moreover, there is a 58% chance of performance degradation. This indicates that it is hard to predict the impact of upgrading the WordPress Docker image from one major version to the next one, based on the WordPress version alone.

Summary of RQ3.1

The wide variation in relative response time improvements for the studied upgrades indicate that it is hard to predict how performance will be affected based on the WordPress version in the image alone. This implies that the performance of WordPress is mostly driven by other components in the image.

3.5.2 RQ3.2: What is the Relation Between the Performance of the WordPress Application and Updates of its Dependencies?

Motivation: The two major dependencies of WordPress are the Apache web server and PHP. As we observed in RQ3.1, upgrading the Docker image of WordPress may change these dependencies as well, which could in turn affect the performance. In this RQ, we analyze the impact of changes in the Apache and PHP versions on the response time of WordPress.

Approach: We grouped the performance measurements of images that use the same patch, minor or major versions of PHP or Apache, as specified in Section 3.5.1. For example, we grouped the average response time of each Docker image of WordPress with PHP major version 5, minor version 6, and patch version 33 (5.6.33) in the PHP 5.6.33 patch group. We analyze these groups for both dependencies in a similar fashion as to what was done for WordPress in Section 3.5.1. Note that in our case study, we have no minor and major version analysis for Apache due to the used Apache versions.

Patch Version Analysis of PHP

Upgrading to the next available PHP patch version degraded WordPress' performance in 71 (58%) out of 123 cases. The last box plot in Fig. 3.7 shows the distribution of the relative average response time improvements for all consecutive patch to patch upgrades of PHP. As the box plot shows, 58% of the time, upgrading to the next patch version of PHP degraded WordPress' performance. The most severe degradation after an upgrade was 25%, while the best upgrade resulted in 80% improvement on the average response time.

Upgrading the last patch version in a minor/major version to the first available patch version in a minor/major version always improved the performance of WordPress. Fig. 3.7 shows the distribution of the relative average response times grouped by PHP patch version. For example, 5.6.32 shows the relative

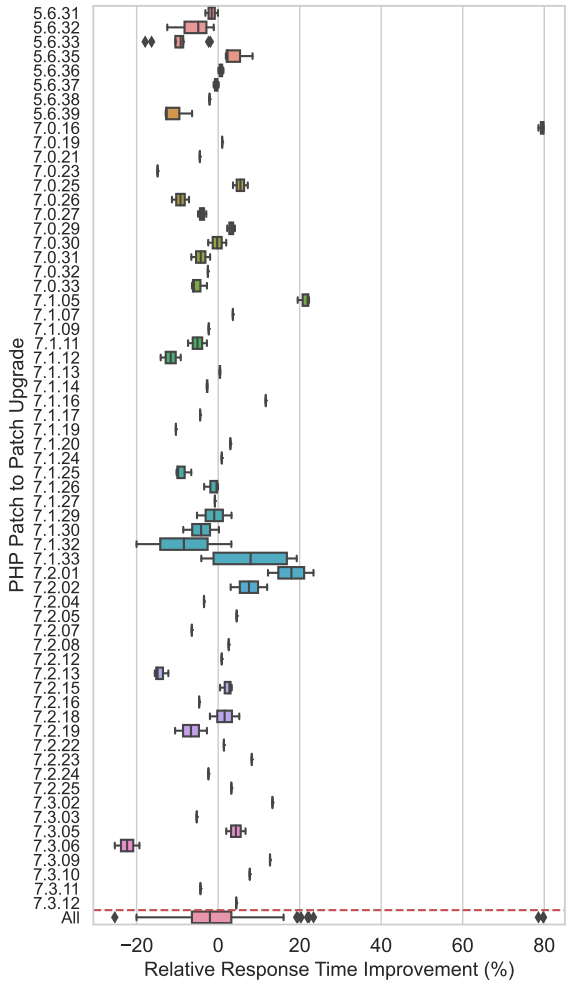


Figure 3.7: Distribution of relative response time improvements for upgrading PHP to the next available patch version. The last box plot shows the distribution of all relative average response improvements for all consecutive PHP patch to patch upgrades.

average response time improvements for upgrading all WordPress Docker images with PHP version 5.6.31 to all WordPress Docker images with PHP version 5.6.32. Fig. 3.7 shows that performing a patch to patch upgrade to the first patch version in a newer minor or major version always resulted in a performance improvement. The highest impact is when upgrading Docker images from PHP patch version 5.6.39 to 7.0.19, resulting in 79-80% improvement. Similar behaviour is observed when upgrading WordPress Docker images with PHP version 7.0.33 to 7.1.05, 7.1.33 to 7.2.01 and 7.2.25 to 7.3.02, boosting the performance by 20-22%, 20-23%, and 13%.

Performing a patch to patch upgrade of PHP within a minor version resulted in an unpredictable impact on the performance of the application. Fig. 3.7 shows that upgrading WordPress Docker images from 5.6.31 to 5.6.32 caused a performance degradation of 1% to 12%, which means that upgrading a WordPress Docker image with PHP version 5.6.31 to the next PHP patch version worsened the performance in all cases. On the contrary, when upgrading from PHP version 7.2.01 to 7.2.02, there was a performance improvement from 3% to 12%. While these improvements and degradations may look small, they can quickly add up when conducting several upgrades in succession. For example, one may do four consecutive patch to patch upgrades from version 7.0.29 to 7.0.33, which could result in a cumulative impact of 14% performance degradation.

Minor Version Analysis of PHP

Upgrading the minor version of PHP in WordPress Docker images resulted in an improved performance in 970 out of 1,474 (66%) cases. The last box plot in Fig. 3.8 depicts the distribution of relative average response time improvements when upgrading all WordPress Docker images with the same PHP minor version to the next minor version (1,474 possible upgrades). In 970 upgrades, we achieved a better average response time. Comparing this finding with the patch to patch upgrade results, we observe that doing a minor to minor upgrade is 58% more likely

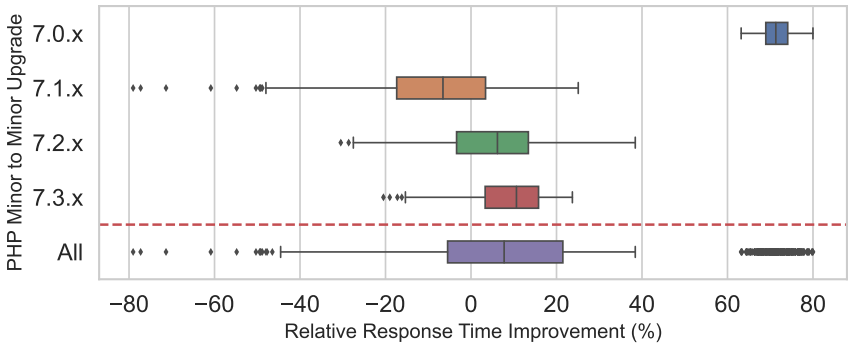


Figure 3.8: Distribution of relative response time improvements for a minor to minor upgrade for different minor versions of PHP. The last box plot shows the distribution of relative response time improvements for all consecutive minor to minor upgrades of PHP in WordPress Docker images.

to result in performance improvement than a patch to patch one. The Mann-Whitney U test confirms that this difference is statistically significant, with a small effect size. Therefore, one who is doing a minor to minor upgrade on the PHP version of the WordPress Docker is less in risk of deteriorating the performance.

In all studied WordPress Docker images, upgrading an image with PHP minor version 5.6 to one with PHP 7.0 improved the performance. Fig. 3.8 shows that for all 306 possible upgrades from PHP version 5.6.x to 7.0.x, an improvement from 63% to 80% was observed for the average response time. This finding indicates that older minor versions of PHP suffered from a relatively bad performance. Taken into account our other observations, we can conclude that the main contributing factor to the performance of WordPress images that use PHP 5.6.x was the PHP dependency.

Major Version Analysis of PHP

Performing a major to major upgrade of PHP improved the performance of the WordPress application in all 1,241 cases. Fig. 3.9 shows the distribution of the relative average response time improvements for all consecutive major to major upgrades of PHP in WordPress Docker images. Based on the studied WordPress Docker images, in all cases, this upgrade had a positive impact on the performance. In

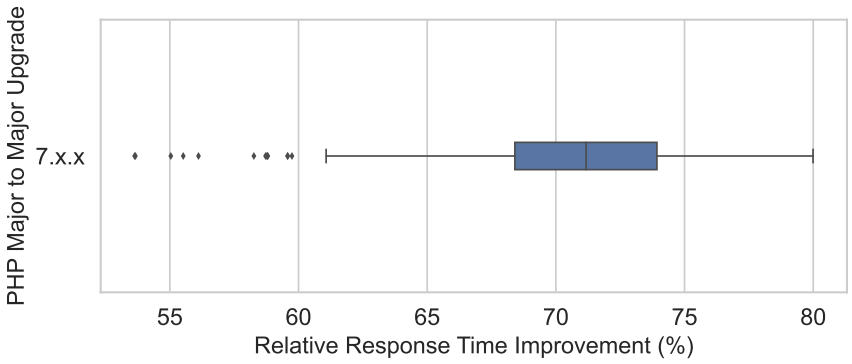


Figure 3.9: The distribution of the relative response time improvements for all consecutive major to major upgrades of PHP in WordPress Docker images.

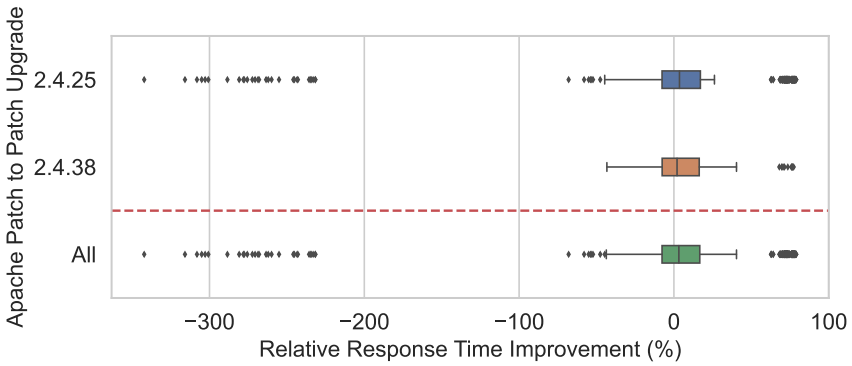


Figure 3.10: Distribution of relative response time improvements for upgrading an Apache patch version to the next available patch version. The last box plot shows the distribution of the average response times of WordPress Docker images with different patch versions of Apache.

other words, selecting a random WordPress Docker image that uses PHP version 5.x.x and upgrading that to another WordPress Docker image with PHP major version 7.x.x resulted in performance improvements ranging from 54% to 80%. This observation confirms our previous observations and highlights how dependent the performance of WordPress is on PHP.

Patch Version Analysis of the Apache Web Server

Upgrading to the next available Apache patch version improved WordPress' performance in 1,183 (58%) out of 2,024 cases. Fig. 3.10 shows the distribution of relative average response time improvements for all patch to patch upgrades of

WordPress Docker images with the same Apache patch version to the next patch version. Fig. 3.10 shows that the chance of performance degradation is 42%, indicating that it is more likely that a patch to patch upgrade of Apache version improves the performance. However, these upgrades can result in 342% performance degradation or 79% performance improvement, such as upgrading from Apache 2.4.10 to 2.4.25. Together with our findings in the PHP analysis, we can conclude that it is hard to predict how upgrading an image will affect WordPress performance, based on its Apache version.

Summary of RQ3.2

It is hard to predict how upgrading a WordPress image will change the performance, based on its WordPress or Apache version. A major version upgrade of the PHP considerably improved WordPress' performance in all cases, implying that WordPress' performance is highly dependent on the PHP version. Upgrading the last patch version in a minor/major version to the first available patch version in a minor/major version of PHP always improved WordPress' performance.

3.6 Discussion

In our case study, we demonstrated how our methodology can help to analyze the performance risks of Docker image upgrades. Ideally, the performance of an image is thoroughly tested before it is being upgraded to. However, performance tests are not popular among developers [33, 135], and they are hard to execute correctly. Hence, our expectation is that many users of images from Docker Hub will conduct an image upgrade without doing such performance tests. The goal of our methodology is to give such users insights about the performance change of doing an image upgrade without conducting a performance test. Below, we discuss the caveats of our methodology.

Caveat 1: The recommendations apply to the studied application only. While we were able to extract several recommendations for the WordPress application (e.g., upgrades of images with PHP 5.x.x to PHP 7.x.x always resulted in improved

performance), these recommendations apply to the WordPress application only. Until there is a large body of performance measurements available for Docker Hub images of many applications, the recommendations of our approach can be applied to the studied application only. Our vision is that through community effort (e.g., as described in Recommendation 1 below), or a future large-scale study, broader-ranging recommendations can be given.

Caveat 2: Collecting the performance measurements of an application takes a considerable amount of time. While the analysis step of our methodology is lightweight in terms of computation, collecting the performance measurements takes a relatively long time. The (short) performance tests that were conducted during our case study took a total of approximately 3 days to complete. For longer performance tests, or a larger number of Docker images, this can be problematic. One could argue whether doing such analysis is beneficial as compared to simply testing the performance of the target image. However, as said above, it is important to keep in mind that many developers and practitioners prefer to avoid performance tests.

In addition, we make the following recommendations.

Recommendation 1: Docker Hub should allow users to provide performance measurements of an image. During our case study, we observed that none of the official WordPress images mentioned anything about performance or response time. As a result, users of such images have to either resort to other sources of information, conduct their own performance tests, or simply take a gamble in terms of performance when doing an image upgrade. We recommend that Docker Hub provides a mechanism for submitting the results of performance tests of the offered images. The analysis done through our methodology (e.g., the patch/minor/major version analysis) can then offer insights for users who wonder about the performance changes of upgrading to an image that does not have performance measurements yet.

Recommendation 2: Semantic versioning should be extended to cover performance changes. At the time of writing, the types of changes that are allowed

by the semantic versioning principle do not cover performance. However, as mentioned before, performance is increasingly important for software and is considered 'the new correct' [87]. Hence, it is a missed opportunity that changes to performance are not covered by semantic versioning. During our case study, we observed that some recommendations can be made for the WordPress application based on the versioning of its dependencies (in particular, PHP). However, we expect that such recommendations would be much stronger when supported by the official semantic versioning specification.

Recommendation 3: Researchers of Docker Hub images must be aware of the relatively large number of convenience tags. During our case study, we were surprised by a large number of convenience tags in the official WordPress repository on Docker Hub. We highly recommend that researchers of Docker Hub images filter such tags as they point to images that already exist in the data set under their original tag, and can bias the analysis.

Recommendation 4: Developers of performance - sensitive Docker images should include an easy way to test the performance of the image. As said above, performance testing is not a popular task, and designing and conducting such tests is complicated. We recommend that developers of Docker images include an easy way to test the performance of the image, for example, a performance test of which the execution steps do not change across images. Such an inclusion would benefit practitioners who wish to conduct a performance test, but are not sure how to. In addition, it would benefit a community effort for collecting performance measurements as suggested in Recommendation 1.

3.7 Threats to Validity

Internal Validity. Threats to internal validity are related to bias and errors in the experimental procedure. One threat is the short period of load testing for each image. We emphasize that the goal of this research study is not to provide a deep analysis

of the performance of WordPress. Instead, we demonstrate through a case study on WordPress how our methodology can be used to study the performance risks of upgrading Docker images of applications with several dependencies.

Another threat to the internal validity of our findings is the variability of measurements that are done within a cloud environment. To investigate the impact of this variability, we tested the performance of the images that had convenience tags and studied the variation between the repeated measurements. For example, if an image had its original tag and three convenience tags, this allowed us to collect four times three repetitions of the performance test of the same image. We found that 95% of the duplicate images have less than $\sim$ 40 ms standard deviation, which is small given that the average response times are roughly between 400 and 1,000 ms. This shows that the variations in the average response times of the duplicate images are negligible and that the selected cloud environment did not impact the performance test measurements.

Another threat to the internal validity of our findings is that some of the studied upgrades are unlikely to be conducted in the real world. For example, it may be unlikely that users of an image with WordPress 4.x.x and PHP 7.x.x upgrade to an image with WordPress 5.x.x and PHP 5.x.x. However, it is important to study such upgrades as well for two reasons: (1) users may have a very specific reason for doing this upgrade (e.g., avoiding a vulnerability in PHP 7.x.x), and (2) users may not be aware that they are also changing the Apache version. While this change is described quite clearly for WordPress, dependency changes for other applications are much less obvious yet plentiful [80].

External Validity. Threats to external validity question the extent of generalizability of our findings. In this study, we proposed a methodology to study the performance risks of upgrading the Docker Hub image of an application. The findings for the specific dependency versions apply to WordPress and its two main dependencies, PHP and Apache only. However, our methodology is agnostic to the studied

application and dependencies, and can easily be applied to study the performance risks of other applications on Docker Hub.

In this study, we relied solely on the average response time of the application to draw conclusions about its performance. Many other performance metrics exist that can be relevant to capture the performance of an application. While our methodology is agnostic to the studied performance metric, future studies should further investigate the effectiveness of our methodology to study other metrics.

Construct Validity. Threats to construct validity show how well a test can measure what it is supposed to measure. In our case study, we used the simplest performance test available for a webpage: loading its main page. Our case study findings show that even for such a simple performance test, dependencies can already strongly influence the performance of the main application. As our methodology is agnostic to the type of performance test that is executed, we invite researchers to conduct more thorough performance tests for WordPress, but also for other applications (as explained in Section 3.6).

Conclusion Validity. Threats to conclusion validity are concerned with issues that may affect the ability to draw conclusions regarding the experiment setting and the results [234]. To ensure the validity of our approach and the data, we provide a publicly available repository of the automated performance testing tool along with the list of studied WordPress Docker images and a summary of the results [12].

3.8 Conclusion

In this chapter, we propose a methodology to study the performance risks of upgrading Docker Hub images. We demonstrate our methodology through a case study on 90 official Docker Hub images of the WordPress application. In particular, we show how conducting an analysis at the patch, minor and major level can reveal how the performance of an application is tied to its dependencies, rather than to the main application itself. We make the following contributions:

- The first methodology for studying the performance risks of upgrading Docker Hub images.
- A case study on WordPress that shows that there are considerable variations in the performance of Docker Hub images with the same version of the WordPress application (yet different dependency versions), highlighting the impact of dependencies on the performance of WordPress.

Our methodology can be beneficial for practitioners who wish to be informed about the change in performance they can expect when upgrading a Docker Hub image, without conducting a performance test of that image themselves. In particular, we call upon the community to start collecting performance measurements for Docker Hub images of a wide range of applications. These measurements can then be collected into a performance repository, which in turn can be leveraged by our methodology to provide recommendations about the expected performance of Docker images for which no performance measurements exist yet.

Chapter 4

Micro-FL: A Fault-Tolerant Scalable Microservice-Based Federated Learning Platform

4.1 Abstract

As the number of applications of machine learning (ML) increases, rising data privacy concerns expose the limitations of traditional cloud-based ML methods that depend on centralized data collection and processing. Federated learning emerges as a promising alternative, providing a novel approach to training machine learning models that protects data privacy. Federated learning facilitates collaborative model training across different entities, with each user training models locally and only sharing the local model parameters with a central server, which then generates a global model based on these individual updates. This approach ensures data privacy as the training data itself is never directly shared with a central entity.

However, existing federated ML frameworks are not without challenges. Considering the federated learning server design, these frameworks exhibit limited scalability as the number of clients increase and are highly vulnerable to system faults, particularly as the central server becomes a single point of failure. In this chapter we introduce `Micro-FL`, a federated learning framework that uses a microservices architecture to implement the federated learning system. We show that the framework is fault-tolerant

and can be scaled to handle increases in the number of clients. Our comprehensive performance evaluation confirms that `Micro-FL` proficiently handles component faults, thereby enabling smooth and uninterrupted operation.

4.2 Introduction

Over the last decade, the rapid progression of machine learning technologies has propelled a wave of artificial intelligence applications, encompassing fields such as computer vision, anomaly detection, fault diagnosis, and natural language processing. The ascendency of machine learning is largely attributable to two key factors: the accessibility of vast volumes of data and significant advancements in computational techniques and resources.

Nevertheless, the availability of extensive data, metaphorically a "double-edged sword" [142], poses significant personal information leakage risks when customer, industrial, or public data are not appropriately managed and utilized. As an illustration, stringent regulations such as the European Union's General Data Protection Requirements (GDPR) [188] and the United States' California Consumer Privacy Act (CCPA) [44] have been put in place to enhance the protection of personal data and privacy by regulating corporate behaviour [244].

With the escalating focus on data privacy, ownership, and confidentiality in contemporary society, there is a growing apprehension that personal information could be exploited for commercial or political motives without the individual's consent. This concern has catalyzed the emergence of a novel era in machine learning, characterized by approaches specifically designed to safeguard user data privacy. An example of such an approach is Federated Learning (FL), a technique introduced by McMahan et al. [156].

Federated learning serves as a privacy-focused alternative to machine learning approaches that require central data collection, allowing models to be trained directly at the data storage site of each user. This approach eliminates the need for data

transmission, as only the locally trained model parameters are used to develop and refine a more effective global model.

Federated learning systems, depending on the communication scheme between components, can be implemented in either centralized (client-server) or decentralized (peer-to-peer) fashion [104, 138]. In a centralized scheme, the central server primarily orchestrates the training process and sets up the communication infrastructure among users. However, its pivotal role also introduces a potential vulnerability, rendering it a single point of failure within the system. Conversely, in a decentralized scheme, all clients can autonomously coordinate to acquire the global model, facilitating model updates and aggregations via peer-to-peer client interactions.

Although decentralized federated learning methods, such as those based on blockchain [50, 51, 119, 127, 229, 230], can mitigate the challenges of centralized federated learning by eliminating the central server, they introduce their own challenges [148, 214]. These include performance degradation, as well as increased computational and storage costs. Consequently, in this study, we will focus on federated learning systems that employ a centralized communication scheme and tackle its specific challenges.

In a centralized federated learning system, a central server might become a vulnerability, acting as a single point of failure due to physical damage, server node failure, or network disruptions. This can potentially interrupt the federated learning process. While large organizations may handle such server roles in some scenarios, collaborative learning often faces constraints regarding the availability and reliability of a robust central server [222]. The server may also become a bottleneck when serving numerous clients, as highlighted by Lian et al. [143].

Hence, when conceptualizing a federated learning system based on a centralized design, it is essential to adopt a design pattern that is both fault-tolerant and performant. Even though numerous platforms and frameworks exist for federated learning, challenges related to performance, scalability, and fault tolerance remain. While these platforms often emphasize user scalability and fault management at the user-end,

they frequently neglect the vital aspect of server-side fault management and system scalability as the user base expands. Ideally, a federated learning system should inherently possess scalability and fault tolerance.

Scalability in the context of federated learning denotes the ability of the system to incorporate additional devices in the federated learning process. More devices can improve the accuracy and speed up the convergence of federated learning process [77, 211]. Techniques such as resource optimization, prioritizing devices with high computational power, and implementing compression schemes for learning model parameter transfer can aid scalability. Effective resource optimization allows more devices to partake in the federated learning process, thereby enhancing performance. However, increasing device participation necessitates an expansion of server-side computational resources.

Fault-tolerance in the context of federated learning indicates the system's capability to manage the federated learning process effectively, even when the server fails. Traditional federated learning, relying on a centralized cloud server for global aggregation, can be disrupted if the aggregation server malfunctions [109]. Current concerns about fault tolerance in federated learning systems often revolve around adversarial or Byzantine attacks targeting the central server [34, 203, 217]. While numerous studies have investigated system performance, research into the effects of server faults, such as physical damage, on the federated learning system's performance is relatively scant.

To fulfill these properties for a performant federated learning system, we present `Micro-FL` in this chapter. `Micro-FL` is a microservices-based federated learning platform engineered to handle an expanding user base, guarantee high availability, and offer fault tolerance. Suitable for deployment either on-site or in the cloud, it harnesses the flexibility, modularity, scalability, and reliability that microservices provide. `Micro-FL` streamlines the testing, deployment, and maintenance of federated learning algorithms, while allowing dynamic resource allocation based on workload or

user count, thus improving the resource efficiency.

4.3 Background

This section describes the fundamental concepts leveraged in this study, specifically focusing on federated learning and microserives.

4.3.1 Federated Learning

Federated Learning, introduced by Google [156], is a distributed machine learning strategy focused on data privacy. It brings together numerous clients, such as edge devices and organizations, to collaboratively train a shared statistical model, known as a global model. The process, facilitated by a central server, occurs across remote client devices without directly sharing data.

Federated learning is characterized by two main features: 1) It involves a multi-party collaboration, with at least two entities, to construct a machine learning model. Each participant holds unique data that contributes to model training. 2) During the training process, each party's data is kept localized and is not transferred elsewhere.

Federated learning can be formulated as follows: consider a scenario with N clients, denoted as $\{F\}_i^N$, each possessing unique datasets $\{D_i\}_i^N$. In traditional machine learning, these datasets $\{D_i\}_i^N$ would be sent to a central server to train the unified model M_{SUM}. However, this process requires each client F_i to disclose its dataset D_i to the central server, which poses potential data leakage risks.

On the other hand, in federated learning, clients work together to train a model M_{FED} without needing to share their respective datasets $\{D_i\}_i^N$. Suppose V_{SUM} and V_{FED} represent the performance metrics (such as accuracy, recall, or F1-score) of the traditional machine learning model M_{SUM} and the federated model M_{FED}, respectively. If we denote δ as a non-negative real number, we can say that the federated learning model M_{FED} experiences δ-performance loss if:

$$|V_{\text{SUM}} - V_{\text{FED}}| < \delta. \tag{4.1}$$

Equation 4.1 implies that while federated learning builds a machine learning model using decentralized data sources, the model's performance on unseen data closely matches that of a model built on centrally collected data [244].

The federated learning training process can be broadly classified into three steps, as delineated by Lim et al. [145]. The model trained at each client's end is termed the local model, whereas the model synthesized by the federated learning server is denoted as the global model. Figure 4.1 provides a visual representation of the federated learning architecture and its training process.

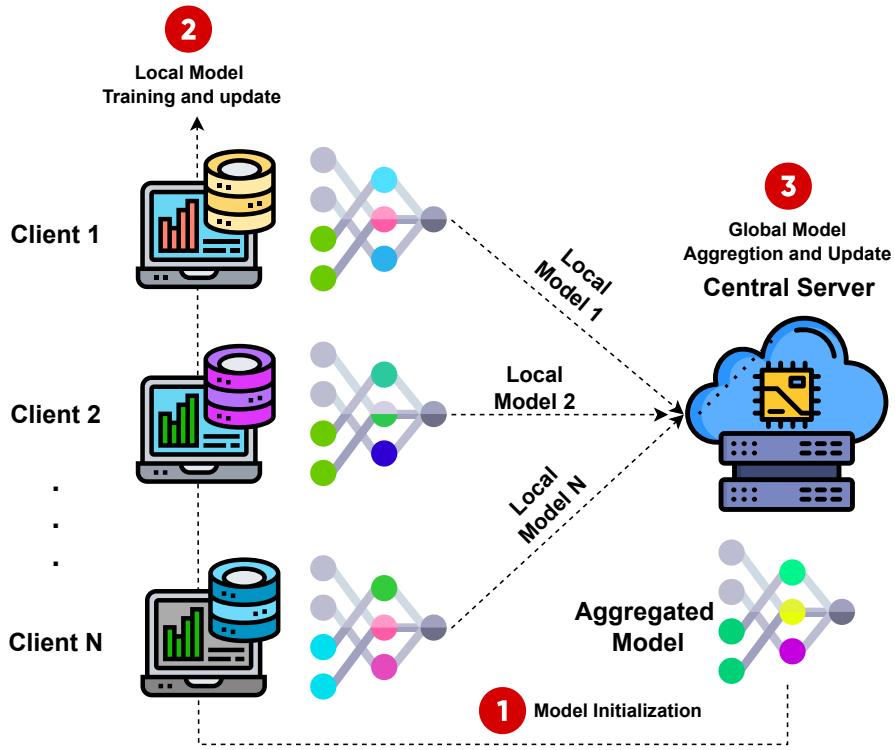


Figure 4.1: An example of a federated learning architecture: client-server model.

1. *Initialization*: The server defines the machine learning task, data prerequisites, and training hyperparameters. It then broadcasts the initial global model parameters w_G^0 to the selected clients [56, 173, 227].

2. *Local model training and update*: Each client i downloads the broadcasted model parameters w_G^t from the server, for iteration number t. They train the model on their local data to obtain updated model parameters, w_i^t, which minimize a specific loss function, $L(w_i^t)$. These updated model parameters are then sent back to the server.

3. *Global model aggregation and update*: The server aggregates the local models generated by the clients and prepares the model parameters for the subsequent training iteration, w_G^{t+1}, with the aim of minimizing the global loss function, $L(w_G^t)$.

   Steps 2 and 3 are repeated until the global loss function converges or a targeted training accuracy is achieved.

Federated Averaging (FedAvg) is a straightforward and commonly used method for aggregating local models in federated learning, proposed by McMahan et al. [156]. This algorithm, averages the updated weights from each client's local model to create a new global model.

4.3.2 Microservices

Monolithic and microservice-based architectures currently dominate the realm of business application development [74]. The monolithic architecture, a traditional approach, constructs an application as a single extensive codebase or repository that encompasses various services and they are not independently executable [37]. This tightly-coupled architecture operates as a singular process in the application server's environment during request handling, with all internal communications managed by an intra-process mechanism. However, as new features are continually integrated in today's fast-paced development cycle, the growing codebase and complexity make code understanding and modification more challenging [75, 190], leading to slower deployment. Another issue with the monolithic architecture is its lack of fault tolerance.

In other words, there is no provision for a system component to function independently when another component fails [19], which is possible with the microservices-based architecture.

In contrast, microservices, a rising trend in software architecture, emphasize the design and development of software that is highly maintainable and scalable [63]. By functionally breaking down large systems into a collection of independent, smaller systems, microservices help manage the growing complexity of software systems.

Microservices typically employ containerization technologies, such as Docker, which encapsulate each service and run it within a container. This structure allows for effortless scalability with minimal latency and hardware resource footprint. Docker containers, which are lightweight, efficient, and can swiftly scale based on needs [161], prove particularly beneficial for a microservices architecture. For this reason, Docker containers have been utilized for the proposed federated learning platform in this study.

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications [129]. It offers scalability by dynamically adjusting containers based on resource demands. It ensures high availability through automatic restart or rescheduling of failed containers, and promotes portability and prevents vendor lock-in through infrastructure abstraction. Furthermore, it optimizes resource utilization by efficiently scheduling containers.

Hence, integrating a federated learning framework that uses a microservices architecture can facilitate the creation of a performant federated learning system that is both scalable and fault-tolerant.

4.4 Related Work

Addressing the single point of failure and enhancing fault tolerance in centralized federated learning has garnered substantial research attention. One approach is the implementation of a decentralized federated learning design, which eliminates the cen-

tral server from the federated learning system, thus averting any single point of failure. This is made possible through the use of different blockchain technologies [50, 182, 229], such as proof-of-work [119], proof-of-authority [127], and proof-of-contribution [214], in conjunction with smart contracts [230, 239]. Such a setup enables model updates and aggregations via direct client-to-client interactions[51], enhancing the overall robustness and fault-tolerance of the system.

However, blockchain-integrated federated learning systems face several challenges [148, 214], including: 1) Performance issues due to a constrained number of transactions, which can result in high latency; 2) The high computational cost of aggregation processes due to typically limited resources on client devices; 3) Increased storage demands, as machine learning models must be stored on all client devices, leading to considerable strain on storage resources; and 4) Potential data privacy risks as all models are accessible to client devices.

Given these challenges with decentralized federated learning, our research proposes an alternative approach to improve the fault tolerance of centralized federated learning. This approach involves implementing a microservices-based design pattern for federated learning.

The following section examines previous research relevant to our study of a microservice-based platform for federated learning. In particular, we delve into existing tools that use a centralized server design for federated learning and their unique features.

TensorFlow Federated (TFF) [83] is an open-source framework for machine learning on decentralized data. Its interfaces include the high-level Federated Learning API for federated learning training with pre-existing TensorFlow models, and the lower-level Federated Core API for developing new federated learning algorithms. While it supports various aggregation functions, it currently lacks GPU utilization for ML model training and only supports simulation mode. The framework is still under development, and its current limitations suggest it might not be suitable for all use

cases.

Federated AI Technology Enabler (FATE) [67] is an open-source project by Webank's AI Department, designed to provide a secure computational framework for a federated AI ecosystem. FATE offers a suite of features like federated statistics, feature engineering capabilities, machine learning algorithm support, and secure protocols. It can be deployed in simulation or federated modes, with installation streamlined via Docker containers. However, its high resource requirements, including 6GB RAM and 100GB disk space on both client and server-side, may render it impractical for real-world federated learning scenarios.

Paddle Federated Learning (PFL) [20]is an open-source platform that supports both horizontally and vertically partitioned data, and can handle neural networks and linear regression models. It leverages techniques like Federated Averaging, Secure Aggregation, and Differentially Private Stochastic Gradient Descent for model construction. Communication in PFL is managed using the ZeroMQ protocol, and it supports both simulation and federated modes, making it adaptable for various deployment scenarios.

PySyft [175] is an open-source project focusing on secure, private deep learning. It comprises components like PyGrid for connecting data owners and data scientists in a peer-to-peer network, KotlinSyft for training PySyft models on Android, SwiftSyft for iOS, and Syft.js for web interfacing. These elements collectively enable secure, collaborative training of models using PySyft.

The Federated Learning and Differential Privacy (FL&DP) [205] Framework is an open-source framework that uses TensorFlow for deep learning tasks and the SciKit-Learn library for linear models and clustering. It offers various aggregation algorithms for and uses adaptive Differential Privacy and randomized response coins to enhance data privacy protection during the learning process.

LEAF [43] is an open-source benchmark tailored for federated learning settings. It provides open-source datasets suitable for federated learning, metrics for evaluating

federated learning algorithms, and a repository of standard methods such as minibatch Stochastic Gradient Descent and Federated Averaging. Serving as a valuable resource, LEAF aids in benchmarking and comparing federated learning algorithms.

Flower [30] is an open-source framework that supports large-cohort training on edge devices and compute clusters. It offers aggregation methods like SecAgg [40] and SecAgg+ [28], and supports both simulation and federated modes of operation. Notably, Flower is language and machine learning framework-agnostic, ensuring broad compatibility.

Serverless federated learning (FedLess) [84] is a system designed for federated learning on diverse Function-as-a-Service platforms, supporting major commercial FaaS platforms such as AWS Lambda, Google Cloud Functions, Azure Functions, and IBM Cloud Functions. Implemented in Python3, FedLess provides a command-line tool for orchestrating the training process and supports TensorFlow and Keras for deep learning models. Its default federated learning strategy is the FedAvg algorithm, commonly used for model updates aggregation. Other research projects, such as [100], have also adopted serverless design for federated learning.

FedML [89] is an open-source research library and benchmark platform designed to aid the development of Federated Learning algorithms and provide objective performance comparisons. It supports on-device training, distributed computing, and single machine simulation. FedML offers resources such as algorithmic implementations, benchmarks with evaluation metrics, access to real-world datasets, and validated baseline results. It is organized into FedML-API for high-level APIs and FedML-core for low-level APIs, using the Message Passing Interface for system communication. FedML supports various federated learning algorithms including FedAvg, Decentralized FL, Vertical Federated Learning, and Split Learning.

Numerous other tools strive to address scalability issues in federated learning systems, predominantly concerning the scaling in the number client [120, 170, 184, 233, 253]. Despite providing several beneficial features, these platforms fail to implement

an efficient central server design that is scalable and fault tolerant. To mitigate these deficiencies, we introduce `Micro-FL`. Grounded in the principles of microservices system architecture, `Micro-FL`-FL is designed to enhance the scalability and robustness of centralized federated learning systems, effectively addressing these significant gaps in contemporary solutions.

4.5 Micro-FL

Figure 4.2 contrasts the building blocks of a commonly-used federated learning server design (monolithic architecture) with the microservices-based design we propose in this study. In a monolithic federated learning server design, all components (e.g., user interface and communication services) are encapsulated into a single process, utilizing a single database. A fault in any of these components can completely halt the federated learning process. This issue, known as a 'single point of failure' within the server, could cause substantial downtime and undermine system reliability. More importantly, scaling monolithic applications requires scaling the whole application, necessitating a considerable increase in resource requirements.

Conversely, when employing a microservices architecture, these components are decoupled from each another (i.e. isolated), each operating as an individual scalable process (microservice) with its own dedicated database. Furthermore, a communication mechanism is implemented for these services to interact with one another. As these microservices are horizontally scalable, multiple instances of each microservice can be created to enhance the fault tolerance of the federated learning system. We refer to this proposed architectural framework as Microservice-based Federated Learning (`Micro-FL`)[1].

`Micro-FL` accommodates both Linear and Deep Neural Network (DNN) models, leveraging TensorFlow and Keras. Additionally, as a microservices-based application, `Micro-FL` possesses several notable attributes:

[1]https://github.com/MikaelSabuhi/Micro-FL

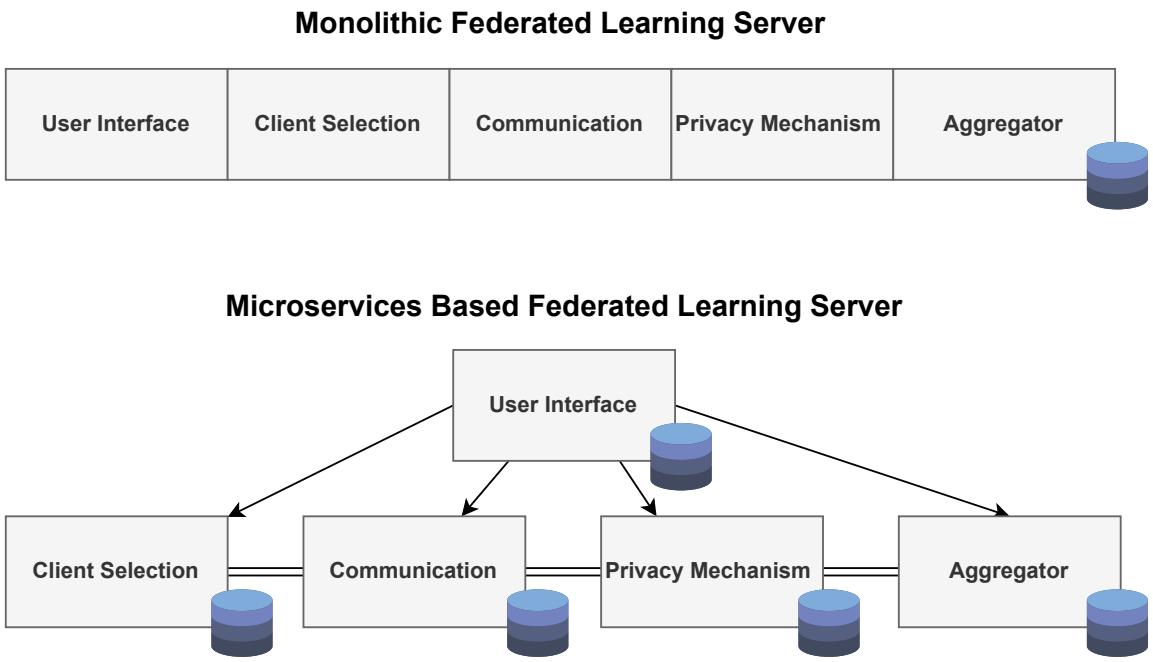


Figure 4.2: Comparison of monolithic and microservices based federated learning systems architectures.

- `Micro-FL`'s distinct modules handle specific functions, contributing to a compact codebase and easier debugging, while also enabling incremental upgrades. These upgrades allow coexistence of old and new versions for compatibility testing, and changes in a module don't require a system-wide reset, thus reducing the re-deployment cycle.

- The framework fault tolerant capabilities of Kubernetes; even with a communication microservice failure, the federated learning process continues without interruption.

- Utilizing containerization, `Micro-FL` allows extensive customization of the deployment environment and facilitates scaling of individual microservices without impacting the whole application. This functionality supports easy deployment or retraction of services based on demand and accommodates both horizontal and vertical scaling.

4.5.1 Micro-FL workflow

The following subsection provides an insight into the workflow of the `Micro-FL` system. The process is characterized by a series of steps that ensure the smooth execution of federated learning tasks. Each of these steps is explained in detail below:

❶ Clients initiate a registration process with `Micro-FL` via a dedicated web application interface. ❷ Based on the number of clients registered and prepared to contribute to the federated learning process, the `Micro-FL` administrator issues a notification signaling the clients regarding the start of a new training iteration. ❸ The Aggregator service actively monitors the connected clients and their statuses. Upon reaching a certain number of participating clients, it triggers the initialization of a model, which is subsequently disseminated to all the clients. ❹ Clients continuously listen for updates from the aggregator service. Upon receipt of the model from the aggregator service, they start training on their local datasets. ❺ Post training, the clients transmit their model parameters to the server through communication service. ❻ All messages submitted by the clients are securely transmitted via the communication service and are logged into the database. ❼ The aggregator microservice constantly monitors client messages during each iteration. When the number of messages equals the total number of clients, the aggregator synthesizes a new global model using the individual client models. ❽ The Aggregator service dispatches a fresh message to the clients, and the cycle from steps ❸ to ❽ repeats. This iterative process continues and is monitored until a specified number of iterations are completed or a pre-defined model performance metric is achieved.

4.5.2 Framework Design

A minimalistic implementation of the proposed `Micro-FL` architectural design is presented in Figure 4.3. All services operate as Docker containers and are orchestrated using Kubernetes. Additionally, load balancing is utilized to distribute clients requests between the user interface and communication microservices. Building upon the

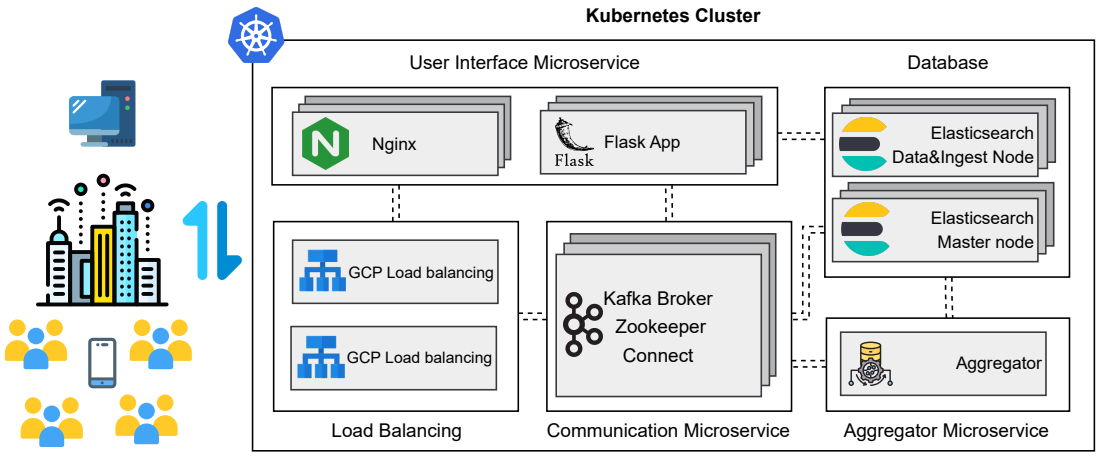


Figure 4.3: Overview of the proposed Micro-FL framework design and its components running on a Kubernetes cluster.

previous section, we discuss the essential components, services, and applications used to actualize `Micro-FL`. We selected these components for their performance, scalability with Kubernetes, and open-source nature. Each microservice is briefly explained as follows:

1. **User Interface.** The web application is developed using the Flask library for Python and Nginx as web server. A federated learning client can register and authenticate using this web UI. Also, Google Cloud Load balancing is used to balance the workload to the web application. Both web application and server are deployed with 3 replicas to improve their performance and reliability.

2. **Database.** The database governs federated learning database access and caching. Elasticsearch (ES), a NoSQL database, retains Federated Learning models, accuracy metrics, and client information. The proposed `Micro-FL` platform employs Elasticsearch due to its scalable and fault-tolerant design, which incorporates index replication and sharding. It uses REST APIs for data storage and search, with a document-based structure in place of tables and schemas. Furthermore, Kibana is utilized for visualizing the federated learning process.

3. **Communication.** This microservice enables data exchange across various

applications, services, and systems, critical for microservices and clients communication. We use Apache Kafka as our message broker, known for its scalability, fault tolerance, and ability to handle trillions of daily messages with minimal latency. Within Kafka, partitions serve as the foundational units for parallelism and scalability. These partitions segment a Kafka topic into multiple smaller, immutable, ordered sequences of records, each hosted on a distinct Kafka broker within a Kafka cluster. The existence of multiple partitions in a Kafka topic paves the way for parallel processing and scalability. Producers can write to various partitions simultaneously, and consumers can consume data from numerous partitions concurrently. This design fosters high throughput and fault tolerance. Kafka employs replication to ensure fault tolerance and high availability. Each partition in a topic is replicated across several brokers to offer redundancy. The replication factor outlines the number of copies of each partition that should be preserved in the cluster. We utilized Strimzi Kafka [103] for Kafka brokers' deployment on Kubernetes and Apache Camel Elasticsearch sink connector (Kafka Connect) for message transfer to the Elasticsearch database.

4. **Aggregator.** Aggregator microservice is responsible for aggregating client updates. It retrieves model updates from the database and creates a new global model for the subsequent federated learning iteration once all chosen users have reported their local model updates. Although numerous methods and structures can be used for aggregation, we opted for the simple and popular FedAvg algorithm in our tests. Since the aggregation happens at the end of the federated learning process and being a synchronous process, we did not replicate this microservice. In other words, in case of fault in this microservice, the Kubernetes controller manager will automatically restart it and it does not impact the training process.

4.6 Evaluation Methodology

In this section, we describe our methodology for evaluating the fault tolerance capabilities of the proposed Micro-FL platform. The working environment will be provisioned utilizing Google Cloud Platform (GCP) and Google Kubernetes Engine. The various steps encompassing our methodology are illustrated in Figure. 4.4. Subsequent subsections will provide an in-depth explanation of each of these steps.

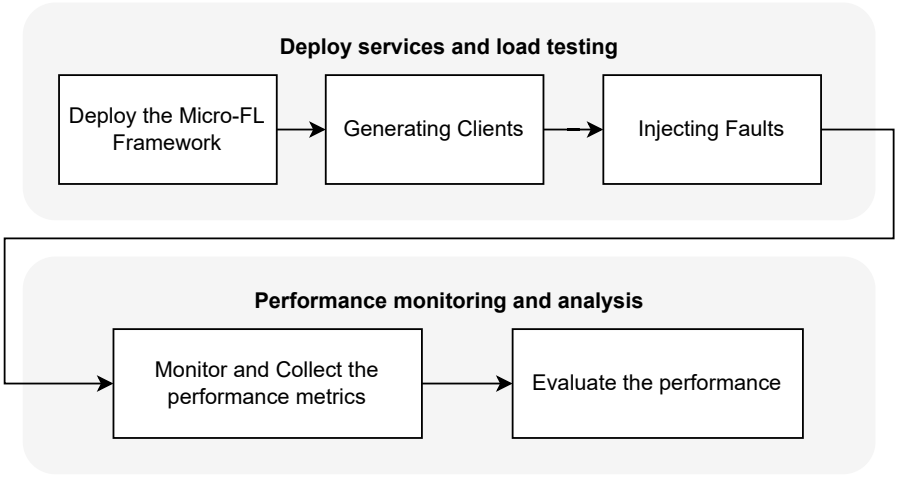


Figure 4.4: Overview of our methodology to evaluate the fault tolerance of `Micro-FL`.

4.6.1 Deploying the Micro-FL framework

The specifics of the `Micro-FL` deployment are detailed in Table 4.2, with the Kubernetes cluster configuration demonstrated in Table 4.1. We used the Google Cloud Kubernetes Engine to deploy the `Micro-FL` frameworks, employing three Kubernetes nodes. This setup will be used to evaluated the fault tolerance behaviour of the microservices, maintaining system robustness in the face of potential faults in the instances.

For the Kafka broker and Kafka connect, we implemented a replication factor of three across all Kafka topics (which is recommended for production level [160]). Simultaneously, the minimum in-sync replicas parameter was configured to two, providing resilience against any single instance failure. We have also established three replicas for the ZooKeeper microservice. In the case of Elasticsearch, indices were

Table 4.1: Configuration of the `Micro-FL` Kubernetes instance.

Property	**Value**
Machine Family	E2-Standard-16
Number of Nodes	3
vCPUs	16
RAM	64GB
Image Type	Ubuntu With Containerd
Boot Disk Type	Balanced Persistent Disk
Boot Disk Size	100GB
Zone	us-central-c
GKE Ver.	1.25.8-gke.500

configured with a replication factor of three. Kibana is utilized for data visualization tasks within the `Micro-FL` platform.

Table 4.2: Micro-FL deployment allocated resources.

Service Name	**vCPU**	**RAM(GB)**	**Replica**	**Ver.**
Kafka	2	8	3	3.4.0
Zookeeper	0.5	4	3	3.7.1
Connect	2	8	3	3.4.0
ES-Master Node	1	4	3	8.7.0
ES-Data/Ingest Node	4	16	3	8.7.0
Kibana	1	4	1	8.7.0
Aggregator	1	4	1	N/A

4.6.2 Generating Clients

Clients are integral to the federated learning process. To evaluate the performance of the proposed Micro-FL system under varying user counts, we simulated clients. Given

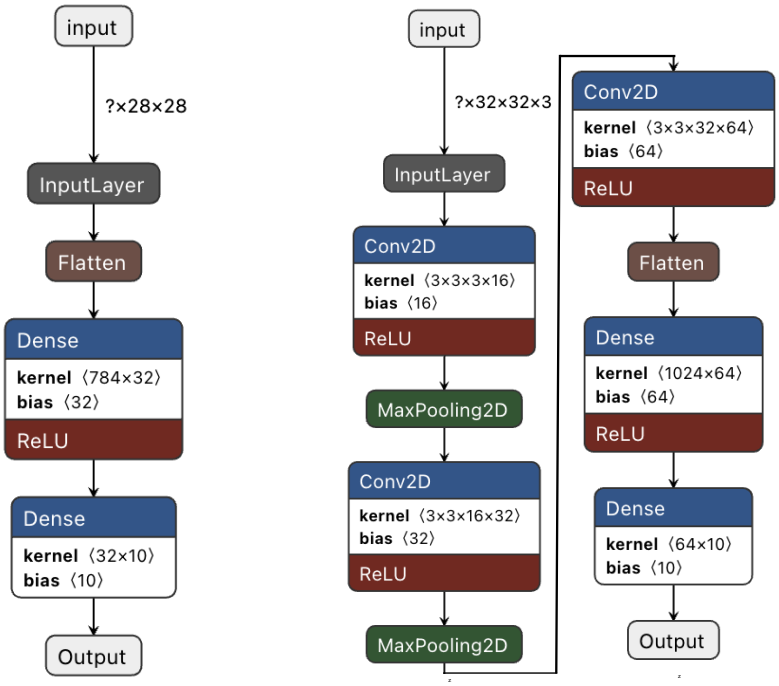
the substantial resources required to run multiple clients concurrently, we utilized Docker containers and Kubernetes for client simulation. Google Cloud Kubernetes Engine is employed to orchestrate variable client counts. The clients contribute to the federated learning process using two popular datasets, MNIST [134] and CIFAR-10 [128], on two distinct models, as shown in Figure 4.5a and Figure 4.5b respectively. The datasets are randomly selected and evenly distributed among the clients and do not have any overlap, as described in Table 4.3. Client-side training employs 5 epochs with a batch size of 10. For optimization, we utilized the Adam optimizer with a learning rate of 0.001 and the training runs for 100 iteration. For conciseness, these datasets and their corresponding models will be referred to as the MNIST and CIFAR-10 workloads. Additionally, the computational resources assigned to each workload are outlined in Table 4.4. It is important to clarify that our primary focus is server-side faults. Hence, we operate under the assumption that clients function as expected, without any faults.

Table 4.3: Distribution of MNIST and CIFAR-10 datasets for different number of users.

Dataset	Clients	Training Samples	Testing Samples
MNIST	100	600	100
	500	120	20
	1000	60	10
CIFAR-10	100	500	100
	500	100	20
	1000	50	10

Table 4.4: Resource allocation for each client for the MNIST and CIFAR-10 datasets.

Dataset	**Instance Type**	$\frac{vCPU}{Client}$	$\frac{RAM}{client}$	**Zone**
MNIST	General Purpose	0.1	0.6 GB	us-central1(a,b,c,f)
CIFAR-10	General Purpose	0.25	1.4 GB	us-central1(a,b,c,f)



(a) Model trained on MNIST. (b) Model trained on CIFAR-10.

Figure 4.5: Trained models for performance analysis of the Micro-FL framework. The model trained on MNIST has 25,450 trainable parameters and for CIFAR-10 it has 89,834.

4.6.3 Injecting Faults

To evaluate the fault tolerance properties of the `Micro-FL` system, we utilized Chaos-Mesh [1], a robust chaos engineering platform specifically designed for Kubernetes. Chaos-Mesh supports a comprehensive array of fault types, including pod and network faults, while also being safe and manageable. For our experiments, we employed Chaos-Mesh version 2.5.2.

We consider the Kafka broker to be the most essential component of the Micro-FL system, as all communications between services and clients pass through it. Message loss may result from a malfunction in this module, severely impeding the federated learning process. Thus, we restrict our fault analysis to Kafka brokers.

We conducted federated learning using the MNIST and CIFAR-10 workloads under two distinct conditions, specifically the healthy and faulty scenarios. Under the healthy scenario, the system operates without any injected faults. In the faulty scenario, a `POD_FAIL` fault is injected every 20 iterations (in 100 iterations) during the federated learning training process and lasts for 5 minutes. This fault is randomly inserted into one of the Kafka brokers. The aim of this experiment is to assess whether the `Micro-FL` system can effectively manage these faults and maintain a seamless operation throughout the process.

4.6.4 Monitoring and Evaluating the Performance

We assessed the performance of `Micro-FL` from two perspectives: 1) Federated learning performance, which encompasses the efficiency of the global model and execution time of the experiments, and 2) Software system performance, which includes CPU utilization and metrics pertinent to the message broker.

Federated Learning Performance Metrics

Performance metrics associated with federated learning are gathered and assessed in the Aggregator microservice and are defined as follows:

- *Global Model Performance:* is measured as training and testing accuracy. These metrics indicate the efficacy of the global model in the federated learning process. The aim is to obtain high performance during training and testing. Given that our datasets are addressing a classification issue, our goal is achieving high training and testing accuracies. Notably, in this study, the aggregator evaluates these accuracies on the entire training and testing datasets, to which it has complete access.

- *Experiment Execution Time:* denotes the time taken to complete each experiment. In assessing the fault tolerance of the proposed architecture, the objective is to maintain consistent execution times for each experiment, regardless of whether the operation conditions are healthy or faulty.

Software Performance Metrics

To collect and evaluate software performance metrics, we utilized Prometheus as our monitoring system and Grafana for data visualization. To assess the messaging system's (Kafka) performance, we leveraged the Kafka exporter, which assists in gathering metrics like CPU utilization and partition statuses of the brokers, as well as their throughput. Simultaneously, we employed cAdvisor metrics for pods and containers along with their associated performance metrics such as CPU utilization. We collect the following software performance metrics:

- *Online Partitions:* are active Kafka partitions, also called leaders, that handle data service. In the leader-follower model, the leader broker manages read/write requests while others replicate its data for high availability and fault tolerance. If a leader fails or a broker goes offline, Kafka automatically assigns a new leader or marks replicas as 'under replicated', respectively.

- *Under Replicated Partitions:* in Kafka have fewer replicas than the set replication factor. Kafka actively manages this by monitoring replication status, electing

new leaders for under-replicated partitions, and initiating replication processes. Once replication is caught up, the partitions become fully replicated again.

- *Partitions at Minimum In-Sync Replicas (ISR):* Kafka ensures data integrity by defining a minimum number of replicas (ISR) that must sync with the leader for successful writes. When a message is sent, it is written to the leader and copied to follower replicas. If enough replicas acknowledge the message to meet the ISR, the write is successful. Our Kafka cluster has an ISR of 2, ensuring that messages are stored in at least two brokers, providing tolerance against a single broker failure. The replication factor and ISR can be customized to meet SLA needs.

- *Offline Partitions:* In Kafka, partitions lacking a leader replica are called offline partitions. They cannot perform read/write operations if all replicas are unavailable or have failed. This can disrupt data availability, hindering data writing and consumption. Our goal is a federated learning platform without offline partitions, ensuring continuous operation and data availability, even during Kafka cluster faults.

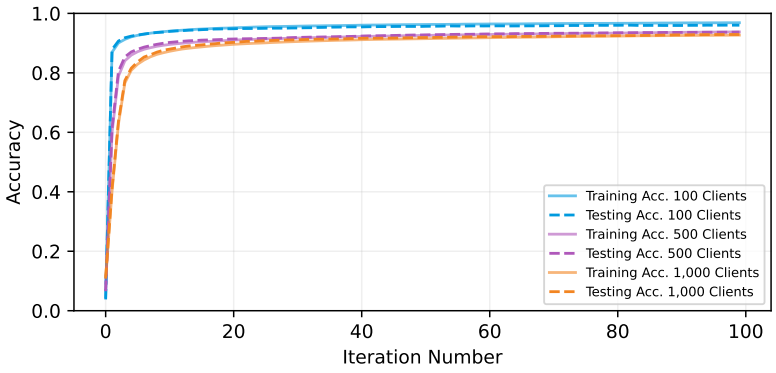
- *Broker CPU Utilization* reflects the proportion of CPU core used by each Kafka broker during the federated learning process. The objective is to utilize allocated resources efficiently, avoiding overutilization.

- *Broker Throughput* signifies the data transmission rate to or from Kafka brokers. As a critical performance indicator of a Kafka cluster, it demonstrates the speed at which producers can relay messages to brokers. High throughput, a core aspect of Kafka's design, enables it to manage real-time processing of substantial, rapid data streams.

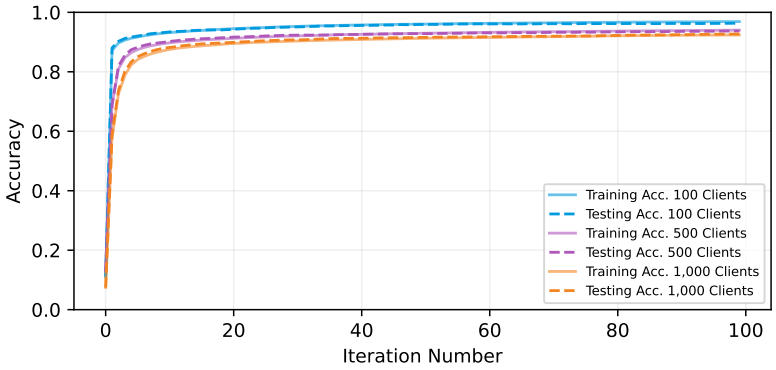
4.7 Results

In this section we will discuss the results of the experiments. We first evaluate the federated learning performance of the `Micro-FL` framework to show that the platform can carry out federated learning with a good model convergence for the MNIST and CIFAR-10 workloads.

4.7.1 Federated Learning Performance



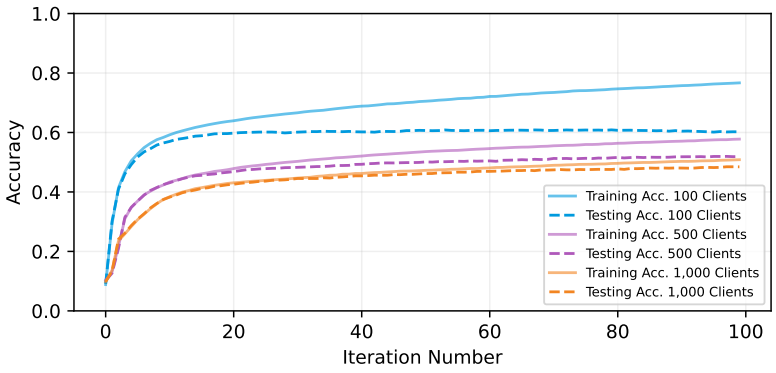
(a) MNIST in the healthy operation scenario.



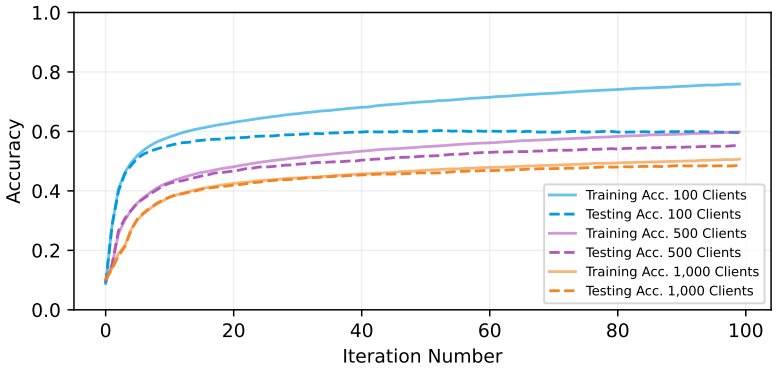
(b) MNIST in the faulty operation scenario.

Figure 4.6: Global model's training and testing accuracies on the MNIST dataset across 100 iterations of training.

The global model of the federated learning system, even under faulty conditions, performs robustly without any significant adverse effects. Figures 4.6 and 4.7 depict the training and testing accuracy of the global model for each



(a) CIFAR-10 in the healthy operation scenario.



(b) CIFAR-10 in the faulty operation scenario.

Figure 4.7: Global model's training and testing accuracies on the CIFAR-10 dataset across 100 iterations of training.

iteration number under healthy and faulty conditions for the workloads discussed earlier. It is evident that faults in the communication microservice do not adversely affect the global model's performance, demonstrating its robust machine learning operation. Table 4.5 details the accuracies achieved for different numbers of users under both healthy and faulty conditions, following 100 iterations of training. These results corroborate the expected performance and uninterrupted convergence of the simple Federated Averaging algorithm, even amidst faults. Minor variations in the training and testing accuracies (such as in the CIFAR-10 workload with 500 clients) are anticipated due to the random dataset sampling.

For optimal machine learning performance, a robust and fault-tolerant federated learning system is necessary, particularly when handling many

clients and large models. In the MNIST workload, an increase in the number of users from 100 to 1,000 induces a minor shift in training accuracy ($\sim 2\%$) and testing accuracy ($\sim 4\%$). However, this impact amplifies for CIFAR-10, resulting in a substantial $\sim 25\%$ decline in both training and testing accuracy, attributed to larger model parameters. Enhanced model performance can be achieved by extending federated aggregation to more iterations and prolonging training durations[77, 211], albeit raising the risk of faults. This observation underscores the need for a robust, long-running federated learning platform. Such a platform should be reliable, fault-tolerant, and capable of integrating an increasing number of users, thereby allowing them to contribute to and improve model performance.

Table 4.5: Training and testing accuracies of the aggregator after training the model for 100 iterations.

Dataset	**#Clients**	**Training Accuracy**		**Testing Accuracy**	
		Healthy	**Faulty**	**Healthy**	**Faulty**
MNIST	100	96.86	96.89	96.02	96.35
	500	94.24	93.26	94.07	93.14
	1000	92.76	92.80	92.83	93.12
CIFAR-10	100	76.69	75.94	60.23	59.60
	500	57.78	59.83	51.90	55.24
	1000	50.84	50.69	48.45	48.58

The proposed Micro-FL platform maintains consistent execution times, with minor fluctuations even under faults, demonstrating its robustness and fault tolerance. Table 4.6 provides the execution times for same workload and healthy and faulty scenarios. From this data, we observe that for the MNIST dataset, there is a minimal variation of less than 3.5% in the experiment execution time. This variation occurred during our experiments with 100 and 500 users on the MNIST dataset, but these fluctuations can be deemed negligible given the short experiment

Table 4.6: Experiment execution time for healthy and faulty scenarios. Positive change indicates shortened execution time and negative shows extended execution time.

Dataset	#Clients	Experiment Duration (s)		Change
		Healthy	Faulty	
MNIST	100	5,230	5,070	-3.06
	500	5,888	6,090	3.43
	1,000	10,231	10,144	-0.85
CIFAR-10	100	11,160	11,014	-1.30
	500	18,819	18,757	-0.33
	1,000	31,256	31,236	-0.06

durations and fluctiations in the cloud infrastructure. For more extensive federated training procedures, such variations become even less pronounced. For instance, with the CIFAR-10 dataset and 1,000 users, the discrepancy between the execution times under healthy and faulty conditions is a mere 0.06%. This observation highlights that the proposed `Micro-FL` platform and its fault-tolerant design does not allow faults to influence the execution time. Therefore, the federated learning process does not experience delays due to faults.

4.7.2 Software Performance Analysis

Micro-FL maintains consistent communication and contribution to federated learning even during faults, demonstrating robust fault tolerance. Figure 4.8 illustrates the partition status during the MNIST experiment with 100 users in the presence of faults. Initially, all partitions are online and accessible. When a fault occurs, the affected Kafka broker's partitions become under-replicated but remain available due to the minimum ISR policy, thus preventing any offline partitions. This reveals the fault-tolerance of `Micro-FL`, which maintains continuous communication and contribution to the federated learning system despite faults. Once the

fault is resolved, the Kafka cluster quickly recovers, restoring the number of online partitions and reducing the under-replicated and minimum ISR partitions to zero, thus reaffirming message availability. This resilient behaviour is consistent across all experiments.

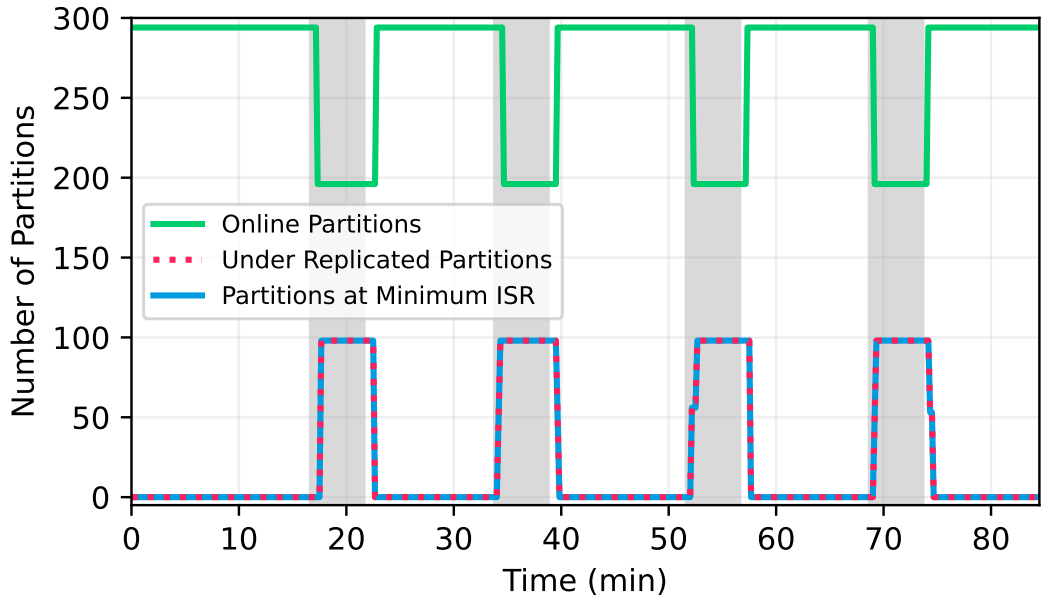


Figure 4.8: Number of online and under replicated partitions and partitions at minimum ISR for Kafka broker during the faulty scenario for MNIST with 100 users. The fault period is marked in gray.

The Micro-FL framework effectively maintains constant throughput and CPU utilization under healthy conditions, and adapts to faults by redistributing load amongst operational Kafka brokers, thereby demonstrating its fault-tolerant nature. Figure 4.9 presents the throughput and CPU utilization under the MNIST workload with 100 users across the three Kafka brokers in the cluster in faulty scenario. In normal operational conditions of `Micro-FL`, the throughput and the CPU utilization of the Kafka brokers remain relatively constant, fluctuating around a specific value. However, according to Figure 4.9, a distinct change is noticeable during a fault occurrence, with darker shades of grey marking the period of time with faulty Kafka broker. Upon fault occurrence, there is a decline in throughput and CPU

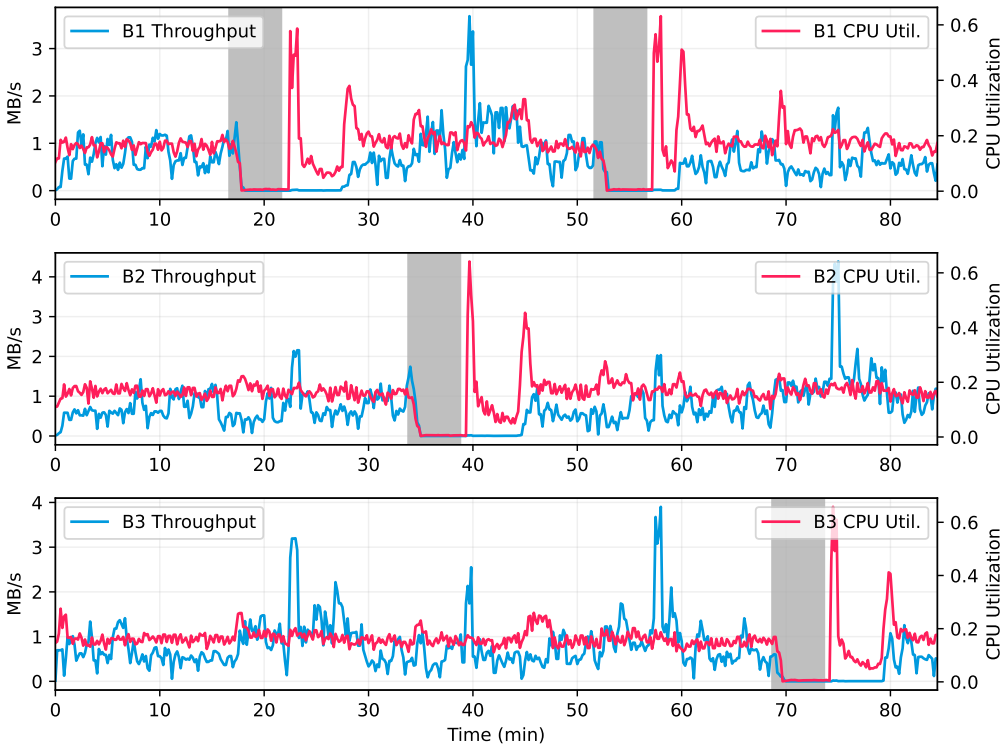


Figure 4.9: Throughput and CPU utilization for all three Kafka brokers in the cluster during the faulty experiment with the MNIST dataset and 100 users. The darker grey denotes the period when the broker is faulty.

utilization for the broker in question. Concurrently, a minor increase is seen in the throughput and CPU utilization of the remaining two operational Kafka brokers. This increase is an adaptive response to compensate for the faulty broker, enabling the system to continue processing messages. Once the fault is resolved, the throughput of the faulty broker remains at zero while the CPU utilization spikes across all Kafka brokers. This is indicative of the faulty Kafka broker retrieving under-replicated partitions from the other brokers.

Micro-FL efficiently manages federated learning with minimal resources utilization, allowing cost-efficient dynamic resource adjustment. As shown in Table 4.7, with an increase in the number of users, the average throughput of brokers linearly increases in both healthy and faulty conditions. The most intensive experiments, involving 1,000 users, show minimal change in average throughput

Table 4.7: Average throughput (MB/s) and CPU core utilization for different experiments.

Metrics	Scenario	MNIST			CIFAR-10		
		100	500	1,000	100	500	1,000
Throughput	Healthy	1.12	5.43	10.28	2.83	17.52	35.59
	Faulty	1.45	6.26	11.06	3.18	17.60	35.76
CPU	Healthy	0.17	0.54	0.74	0.23	0.61	0.76
	Faulty	0.24	0.57	0.85	0.29	0.55	0.87

between healthy and faulty conditions for both MNIST and CIFAR-10 workloads. Despite increasing workload intensity, the rise in CPU utilization is minimal. Even under the most demanding CIFAR-10 workload with 1000 clients, the overall CPU utilization only reaches 38% and 43% in healthy and faulty scenarios respectively. These observations affirm that `Micro-FL` efficiently handles federated learning even with minimal resource allocation, accommodating dynamic resource adjustment for cost-effective system design.

4.8 Conclusion

In this chapter, we highlighted the shortcomings of traditional centralized server designs for federated learning, emphasizing the need for enhanced fault tolerance, scalability, and resource management. We introduced `Micro-FL`, a microservices-based, fault-tolerant system design uniquely created for centralized federated learning setups. Our empirical performance analysis of `Micro-FL`, conducted across varying user counts and two different workloads, showcased its ability to seamlessly manage faults while ensuring an uninterrupted federated learning process. The proposed design facilitates dynamic resource allocation, promoting efficient computational resource management, representing a significant advancement towards more resilient and efficient system designs in federated learning. In future research, we plan to optimize federated learning

system communications using message compression and aggregation techniques that leverage message queuing systems like Kafka to decrease aggregation execution time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In light of recent progress in the field of performant cloud applications, we acknowledge the continued journey toward developing a cloud application that encompasses all core characteristics of performance: reliability, scalability, efficiency, fault tolerance, and responsiveness. This thesis presents three distinct yet interconnected research studies aiming to advance our understanding of strategies to enhance these qualities in the context of performant cloud applications.

In the first research study (Chapter 2), we explored the application of machine learning and adaptive control theory in devising autoscaling techniques. These methods underpin the construction of a cloud software system designed to improve scalability, resource utilization, and responsiveness of the cloud applications. The second research study (Chapter 3) put forward a methodology to investigate the performance risks intrinsic to the process of upgrading Docker images in containerized applications. This research emphasized the critical role of performance testing, shining a light on its significance in the development of performant cloud applications. In our final research study (Chapter 4), we delved into the advantages offered by the microservices architecture in the conceptualization and execution of performant cloud applications. The study accentuates the utility of microservices architecture in the design of cloud-based machine learning platforms, such as federated learning platforms. This serves

to amplify their scalability, reliability, fault tolerance, and efficiency.

Below, we provide a succinct discussion of the methodologies employed, major findings and significant contributions made by each of our research studies:

- **In Chapter 2, we demonstrated how performance modeling of cloud applications can lead to improved scalability by integrating it with a control theoretical autoscaler**. This improvement results from an enhanced understanding of cloud application behavior, including alterations in the cloud infrastructure and workload. To achieve this, we leveraged neural networks as a 'black box' model, capitalizing on their exceptional abilities in learning non-linear functions. Through a comprehensive experimental methodology, we identified the most effective autoscaler system among our adaptive PID controller, its non-adaptive counterpart, and the scaling heat autoscaler. These comparisons aimed at optimizing cloud application performance while maximizing resource utilization and minimizing SLA violations. Our results suggest that the proposed adaptive PID controller offers a robust solution for developing performant cloud software systems.

- **In Chapter 3, we put forth a methodology to investigate the performance risks that arise when upgrading Docker Hub images.** To illustrate this, we used 90 official Docker Hub images of the WordPress application as our case study. Our findings indicate that the performance of an application depends significantly on its dependencies, rather than the application itself. This observation was made possible by conducting an in-depth analysis at the patch, minor, and major levels. Our methodology could benefit practitioners seeking to predict potential performance changes when upgrading a Docker Hub image without the need for individual performance tests. We urge the community to gather performance data for Docker Hub images across a range of applications, which can then be consolidated into a performance repository.

Using our methodology, this repository can then facilitate informed predictions about the performance of Docker images without existing performance measurements. This strategy can guide developers and users of Docker images in designing performant containerized applications, providing a knowledge base for anticipated performance changes in new versions of their applications.

- **In Chapter 4, we discussed the advantages of using microservices architecture in designing cloud-based machine learning platforms, specifically federated learning systems, by proposing Micro-FL.** We drew attention to the limitations of traditional centralized server designs for federated learning, emphasizing the need for improvements in fault tolerance, resource allocation, and scalability. We proposed Micro-FL, a scalable fault tolerant microservices-based system design, tailored for federated learning servers. Our comprehensive empirical performance evaluation of Micro-FL, with various user counts and two distinct workloads, indicated that the proposed platform could efficiently handle faults, thereby ensuring uninterrupted federated learning processes. Moreover, our design supports dynamic resource allocation, leading to more cost-effective management of computational resources. This development marks a significant stride toward more scalable, fault tolerant and efficient system designs in cloud software systems.

5.2 Future Work

While our thesis offers in-depth investigations into several strategies for developing performant cloud software systems, suggesting practical solutions and improvements, there remains a wealth of possibilities for future research extensions. We outline several such research directions below:

- **Examine other machine learning modeling techniques for autoscaler design.** In Chapter 2, we utilized dense neural networks with time delays for

performance modeling of the cloud application. Exploring other methodologies, such as Long-Short-Term-Memory (LSTMs), transformers [223], or Gaussian Process models [226], could potentially enhance the performance modeling stage. Given the stochastic nature of cloud infrastructure, Gaussian Process models may prove particularly effective.

- **Investigate model-free controller design for autoscalers.** Chapter 2 discussed the use of adaptive PID controllers in implementing the autoscaler system. An alternative approach could involve model-free control design[71], which employs an ultra-local equation to design an intelligent PID controller—namely, iPID. However, system identification will remain a necessary component.

- **Explore predictive control design for autoscaler systems.** In Chapter 2, we leveraged adaptive controller design for scaling up/down the cloud application. Predictive controller design [53] could further improve the robustness of the autoscaler, given its capabilities in forecasting the future trend of the cloud application.

- **Integrate actuator (e.g. containers) uncertainty and faults into the fault-tolerant autoscaler design.** Chapter 2 operated under the assumption that containers would deploy readily, without encountering any uncertainties or faults. In reality, containerized applications may experience failures due to internal errors or network issues. Further study into fault tolerance design for autoscalers is necessary, ensuring they can handle application scaling even in the event of container failures.

- **Perform root cause analysis on Docker Images to identify potential incompatibility in application dependencies.** In Chapter 3, our aim was to comprehend the impact of Docker image upgrades on performance and its correlation with dependencies. However, there is a need for methodologies

to identify the root cause of performance degradation, possibly through the application of root cause analysis techniques. Certain application dependencies may prove incompatible and trigger performance degradation, necessitating further investigation.

- **Design tools for automated performance testing of Docker images.** In Chapter 3, we outlined a methodology for performance testing to evaluate the risks of Docker image upgrades. However, our focus was limited to WordPress, a web application. For other applications, such as machine learning applications deployed on the cloud, alternative performance testing approaches are required. We see a need for a tool capable of performance testing a wide variety of containerized applications, with the ability to automatically conduct these tests alongside Docker images. This would allow users to be aware of potential performance risks before upgrading to a newer Docker image version.

- **Explore message compression techniques to enhance the network efficiency of the proposed Micro-FL.** In Chapter 4, we proposed Micro-FL, a federated learning platform using microservices architecture. This architecture adds network overhead to the system, potentially creating a system bottleneck. We suggest that using efficient message compression techniques could significantly reduce network overhead, leading to improved performance.

- **Design a new aggregation algorithm that leverages the queuing system.** In Chapter 4, we utilized the federated averaging algorithm to aggregate the model parameters of the federated learning process. This approach required waiting for all clients to report their parameters to the server, potentially leading to inefficiencies and longer processing times. There is a need for an aggregation algorithm that scales better and can aggregate parameters as soon as they become available on the message broker.

Bibliography

[1] *A powerful chaos engineering platform for kubernetes: Chaos mesh.* [Online]. Available: https://chaos-mesh.org/.

[2] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2018-July, pp. 970–973, 2018, ISSN: 21596190.

[3] A. Abid, M. F. Manzoor, M. S. Farooq, U. Farooq, and M. Hussain, "Challenges and issues of resource allocation techniques in cloud computing.," *KSII Transactions on Internet & Information Systems*, vol. 14, no. 7, 2020.

[4] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang, "Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: An experience report," in *International Conference on Mining Software Repositories (MSR)*, ACM, 2016, pp. 1–12.

[5] A. Al-Said Ahmad and P. Andras, "Scalability analysis comparisons of cloud-based software services," *Journal of Cloud Computing*, vol. 8, no. 1, pp. 1–17, 2019.

[6] A. Y. Alanis, J. D. Rios, J. Rivera, N. Arana-Daniel, and C. Lopez-Franco, "Real-time discrete neural control applied to a Linear Induction Motor," *Neurocomputing*, vol. 164, pp. 240–251, 2015, ISSN: 18728286.

[7] J. P. S. Alcocer, "Tracking down software changes responsible for performance loss," in *Proceedings of the International Workshop on Smalltalk Technologies*, 2012, pp. 1–7.

[8] J. P. S. Alcocer and A. Bergel, "Tracking down performance variation against source code evolution," *ACM SIGPLAN Notices*, vol. 51, no. 2, pp. 129–139, 2015.

[9] H. Alipour and Y. Liu, "Online machine learning for cloud resource provisioning of microservice backend systems," in *2017 IEEE International Conference on Big Data (Big Data)*, IEEE, 2017, pp. 2433–2441.

[10] D. Alshoaibi, K. Hannigan, H. Gupta, and M. W. Mkaouer, "Price: Detection of performance regression introducing code changes using static and dynamic metrics," in *International Symposium on Search Based Software Engineering*, Springer, 2019, pp. 75–88.

[11] K. H. Ang, G. Chong, and Y. Li, "PID control system analysis, design, and technology," *IEEE Transactions on Control Systems Technology*, vol. 13, no. 4, pp. 559–576, 2005, ISSN: 10636536.

[12] Anonymous Publisher, *Docker Image Performance Risk Analysis*, https://github.com/asgaardlab/wip-21-Mikael-DHPanalysis-code, Online; accessed 28 July 2021, 2021.

[13] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE Press, 2017, pp. 64–73.

[14] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva, "Control theory for model-based performance-driven software adaptation," in *2015 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, 2015, pp. 11–20.

[15] D. Arcelli and V. Cortellessa, "Challenges in applying control theory to software performance engineering for adaptive systems," in *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*, Delft, The Netherlands: ACM, 2016, pp. 35–40, ISBN: 978-1-4503-4147-9.

[16] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *Netwrk. Mag. of Global Internetwkg*, vol. 14, no. 3, pp. 30–37, 2000, ISSN: 0890-8044.

[17] M. Armbrust *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[18] T. T. F. Authors, *TensorFlow Federated*, version 0.20.0, Dec. 2018. [Online]. Available: https://github.com/tensorflow/federated.

[19] M. Baboi, A. Iftene, and D. Gîfu, "Dynamic microservices to create scalable and fault tolerance architecture," *Procedia Computer Science*, vol. 159, pp. 1035–1044, 2019.

[20] Baidu, *Baidu paddlepaddle*, http://research.baidu.com.

[21] L. Baresi and S. Guinea, "Event-based multi-level service monitoring," in *2013 IEEE 20th International Conference on Web Services*, 2013, pp. 83–90.

[22] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," pp. 217–228, 2016.

[23] C. Barna, M. Fokaefs, M. Litoiu, M. Shtern, and J. Wigglesworth, "Cloud adaptation with control theory in industrial clouds," in *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, 2016, pp. 231–238.

[24] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "A devops architecture for continuous delivery of containerized big data applications," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, IEEE, 2017.

[25] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering Elastic Containerized Cloud Applications to Enable DevOps," *IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017*, pp. 65–75, 2017.

[26] C. Barna, M. Litoiu, M. Fokaefs, M. Shtern, and J. Wigglesworth, "Runtime performance management for cloud applications with adaptive controllers," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, Berlin, Germany: ACM, 2018, pp. 176–183.

[27] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[28] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly) logarithmic overhead," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1253–1269.

[29] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.

[30] D. J. Beutel *et al.*, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[31] C.-P. Bezemer, E. Milon, A. Zaidman, and J. Pouwelse, "Detecting and analyzing I/O performance regressions," *Journal of Software: Evolution and Process (JSEP)*, vol. 26, no. 12, pp. 1193–1212, 2014.

[32] C.-P. Bezemer, J. Pouwelse, and B. Gregg, "Understanding software performance regressions using differential flame graphs," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 535–539.

[33] C.-P. Bezemer *et al.*, "How is performance addressed in DevOps?" In *ACM/SPEC International Conference on Performance Engineering*, ser. ICPE, 2019, 45–50, ISBN: 9781450362399.

[34] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo, "Analyzing federated learning through an adversarial lens," in *International Conference on Machine Learning*, PMLR, 2019, pp. 634–643.

[35] G. Bhatia, A. Choudhary, and V. Gupta, "The road to Docker: A survey," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 8, 2017.

[36] A. Bhowmick, J. Duchi, J. Freudiger, G. Kapoor, and R. Rogers, "Protection against reconstruction and its applications in private federated learning," *arXiv preprint arXiv:1812.00984*, 2018.

[37] N. Bjørndal *et al.*, "Migration from monolith to microservices: Benchmarking a case study," 2020.

[38] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, 2022.

[39] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service-and microservice-based systems: A literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, 2017, pp. 107–115.

[40] K. Bonawitz *et al.*, "Practical secure aggregation for privacy-preserving machine learning," in *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[41] K. Bonawitz *et al.*, "Towards federated learning at scale: System design," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 374–388, 2019.

[42] T. Bui, "Analysis of Docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[43] S. Caldas *et al.*, "Leaf: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018.

[44] *California privacy rights act: Californians for consumer privacy*, 2021. [Online]. Available: https://www.caprivacy.org/.

[45] N. Carlini, C. Liu, Úlfar Erlingsson, J. Kos, and D. Song, *The secret sharer: Evaluating and testing unintended memorization in neural networks*, 2019. arXiv: 1802.08232 [cs.LG].

[46] E. Casalicchio and V. Perciballi, "Measuring Docker performance: What a mess!!!" In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, pp. 11–16.

[47] A. Celesti, L. Carnevale, A. Galletta, M. Fazio, and M. Villari, "A watchdog service making container-based micro-services reliable in iot clouds," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, IEEE, 2017, pp. 372–378.

[48] M. Chadha, A. Jindal, and M. Gerndt, "Towards federated learning using faas fabric," in *Proceedings of the 2020 sixth international workshop on serverless computing*, 2020, pp. 49–54.

[49] X. Chang, R. Xia, J. K. Muppala, K. S. Trivedi, and J. Liu, "Effective modeling approach for iaas data center performance analysis under heterogeneous workload," *IEEE Transactions on Cloud Computing*, vol. 6, no. 4, pp. 991–1003, 2016.

[50] Y. Chang, C. Fang, W. Sun, *et al.*, "A blockchain-based federated learning method for smart healthcare," *Computational Intelligence and Neuroscience*, vol. 2021, 2021.

[51] Q. Chen, Z. Wang, Y. Zhou, J. Chen, D. Xiao, and X. Lin, "Cfl: Cluster federated learning in large-scale peer-to-peer networks," in *Information Security: 25th International Conference, ISC 2022, Bali, Indonesia, December 18–22, 2022, Proceedings*, Springer, 2022, pp. 464–472.

[52] S. Chen and S. A. Billings, "Neural networks for nonlinear dynamic system modelling and identification," *International Journal of Control*, vol. 56, no. 2, pp. 319–346, 1992, ISSN: 13665820.

[53] W.-H. Chen, D. J. Ballance, P. J. Gawthrop, J. J. Gribble, and J. O'Reilly, "Nonlinear pid predictive controller," *IEE Proceedings-Control Theory and Applications*, vol. 146, no. 6, pp. 603–611, 1999.

[54] L. Cheng, W. Liu, Z. G. Hou, J. Yu, and M. Tan, "Neural-Network-Based Nonlinear Model Predictive Control for Piezoelectric Actuators," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 12, pp. 7717–7727, 2015, ISSN: 02780046.

[55] M. N. Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, "A survey of fault tolerance architecture in cloud computing," *Journal of Network and Computer Applications*, vol. 61, pp. 81–92, 2016.

[56] Y. J. Cho, J. Wang, and G. Joshi, "Client selection in federated learning: Convergence analysis and power-of-choice selection strategies," *arXiv preprint arXiv:2010.01243*, 2020.

[57] T. Combe, A. Martin, and R. Di Pietro, "To Docker or not to Docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, 2016.

[58] C.-C. Crecana and F. Pop, "Monitoring-based auto-scalability across hybrid clouds," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1087–1094.

[59] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[60] D. Desmeurs, C. Klein, A. V. Papadopoulos, and J. Tordsson, "Event-Driven Application Brownout: Reconciling High Utilization and Low Tail Response Times," *International Conference on Cloud and Autonomic Computing*, pp. 1–12, 2015.

[61] Docker, *Create a Docker Swarm manager*, Dec. 2019. [Online]. Available: https://docs.docker.com/swarm/reference/manage/.

[62] Docker, *The world's leading service for finding and sharing container images with your team and the Docker community.* Last accessed 2021-10-12, Dec. 2021. [Online]. Available: https://www.docker.com.

[63] N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[64] J. Durango *et al.*, "Control-theoretical load-balancing for cloud applications with brownout," *Proceedings of the IEEE Conference on Decision and Control*, vol. 2015-Febru, no. February, pp. 5320–5327, 2014, ISSN: 07431546.

[65] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović, and A. van Hoorn, "Microservices: A performance tester's dream or nightmare?" In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020, pp. 1–12.

[66] S. Farokhi, P. Jamshidi, D. Lucanin, and I. Brandic, "Performance-based vertical memory elasticity," in *2015 IEEE International Conference on Autonomic Computing*, 2015, pp. 151–152.

[67] Fate, *An industrial grade federated learning framework*, https://fate.fedai.org.

[68] F. Febrero, C. Calero, and M. Á. Moraga, "Software reliability modeling based on iso/iec square," *Information and Software Technology*, vol. 70, pp. 18–29, 2016.

[69] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *IEEE international symposium on performance analysis of systems and software (ISPASS)*, IEEE, 2015, pp. 171–172.

[70] A. Filieri *et al.*, "Control strategies for self-adaptive software systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 11, no. 4, pp. 1–31, 2017, ISSN: 15564703.

[71] M. Fliess and C. Join, "Model-free control," *International Journal of Control*, vol. 86, no. 12, pp. 2228–2252, 2013.

[72] M. Fokaefs, Y. Rouf, C. Barna, and M. Litoiu, "Evaluating adaptation methods for cloud applications: An empirical study," in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 632–639.

[73] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *10th International Conference on Quality Software*, IEEE, 2010, pp. 32–41.

[74] M. Fowler, *Microservicepremium*, 2015. [Online]. Available: https://martinfowler.com/bliki/MicroservicePremium.html.

[75] M. Fowler, *Microservices*, 2015. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[76] M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, and F. Montesi, "Self-reconfiguring microservices," in *Theory and Practice of Formal Methods*, Springer, 2016, pp. 194–210.

[77] W. Gao, Z. Zhao, G. Min, Q. Ni, and Y. Jiang, "Resource allocation for latency-aware federated learning in industrial internet of things," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 12, pp. 8505–8513, 2021.

[78] I. Gergin, B. Simmons, and M. Litoiu, "A decentralized autonomic architecture for performance control in the cloud," *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pp. 574–579, 2014.

[79] D. Gesvindr and B. Buhnova, "Performance challenges, current bad practices, and hints in paas cloud application design," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 4, pp. 3–12, 2016.

[80] S. Gholami, H. Khazaei, and C.-P. Bezemer, "Should you upgrade official docker hub images in production environments?" In *ICSE New Ideas and Emerging Results (NIER)*, 2021, pp. 1–5.

[81] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[82] Google, *Benchmarks for Linux VM instances*, Last accessed 2020-09-20, 2020. [Online]. Available: https://cloud.google.com/compute/docs/benchmarks-linux.

[83] Google, *Tensorflow federated*, https://www.tensorflow.org/federated.

[84] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, "Fedless: Secure and scalable federated learning using serverless computing," in *2021 IEEE International Conference on Big Data (Big Data)*, IEEE, 2021, pp. 164–173.

[85] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: A case study in multi-cloud microservices-based applications," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 2956–2983, 2018.

[86] B. Guo, H. Liu, Z. Luo, and F. Wang, "Adaptive PID controller based on BP neural network," *IJCAI International Joint Conference on Artificial Intelligence*, no. 2, pp. 148–150, 2009, ISSN: 10450823.

[87] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2018, pp. 1–23.

[88] S. Haselböck and R. Weinreich, "Decision guidance models for microservice monitoring," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, 2017, pp. 54–61.

[89] C. He *et al.*, "Fedml: A research library and benchmark for federated machine learning," *arXiv preprint arXiv:2007.13518*, 2020.

[90] K. Hornik, M. Stinchcombe, H. White, *et al.*, "Multilayer feedforward networks are universal approximators.," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[91] C. Hu, J. Jiang, and Z. Wang, "Decentralized federated learning: A segmented gossip approach," *arXiv preprint arXiv:1908.07782*, 2019.

[92] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 60–71.

[93] IBM Cloud Education, *Containerization*, Last accessed 2021-10-12, Aug. 2020. [Online]. Available: https://www.ibm.com/cloud/learn/containerization.

[94] E. Incerto, M. Tribastone, and C. Trubiani, "Combined Vertical and Horizontal Autoscaling Through Model Predictive Control," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11014 LNCS, pp. 147–159, 2018, ISSN: 16113349.

[95] E. Incerto, M. Tribastone, and C. Trubiani, "Software performance self-adaptation through efficient model predictive control," *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 485–496, 2017.

[96] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *European Conference on Parallel Processing*, Springer, 2005, pp. 196–205.

[97] "ISO / IEC 25010 : 2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," 2013.

[98] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada, "Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures," in *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, IEEE, 2016, pp. 70–79.

[99] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon 2016*, IEEE, 2016, pp. 1–5.

[100] K. Jayaram, V. Muthusamy, G. Thomas, A. Verma, and M. Purcell, "Lambda fl: Serverless aggregation for federated learning," in *International Workshop on Trustable, Verifiable and Auditable Federated Learning*, 2022, p. 9.

[101] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[102] W. John, F. Moradi, B. Pechenot, and P. Sköldström, "Meeting the observability challenges for vnfs in 5g systems," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, IEEE, 2017, pp. 1127–1130.

[103] *Kafka on kubernetes in a few minutes*. [Online]. Available: https://strimzi.io/.

[104] P. Kairouz *et al.*, "Advances and open problems in federated learning," *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021.

[105] G. Kakivaya *et al.*, "Service fabric: A distributed platform for building microservices in the cloud," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[106] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2016, pp. 202–211.

[107] J. Kang, Z. Xiong, D. Niyato, Y. Zou, Y. Zhang, and M. Guizani, "Reliable federated learning for mobile networks," *IEEE Wireless Communications*, vol. 27, no. 2, pp. 72–80, 2020.

[108] N. Kerzazi and B. Adams, "Botched releases: Do we need to roll back? empirical study on a commercial web app," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 574–583.

[109] L. U. Khan, W. Saad, Z. Han, E. Hossain, and C. S. Hong, "Federated learning for internet of things: Recent advances, taxonomy, and open challenges," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 3, pp. 1759–1799, 2021.

[110] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, "Savi-iot: A self-managing containerized iot platform," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, IEEE, 2017, pp. 227–234.

[111] H. Khazaei, J. Misic, and V. Misic, "A fine-grained performance model of cloud computing centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2138–2147, 2013.

[112] H. Khazaei, J. Misic, V. Misic, and S. Rashwand, "Analysis of a pool management scheme for cloud computing centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 849–861, 2013.

[113] H. Khazaei, J. Misic, and V. B. Misic, "A fine-grained performance model of cloud computing centers," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 11, pp. 2138–2147, 2012.

[114] H. Khazaei, J. Misic, and V. B. Misic, "Performance analysis of cloud computing centers using m/g/m/m+ r queuing systems," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 5, pp. 936–943, 2011.

[115] H. Khazaei, J. Misic, and V. B. Misic, "Performance of cloud centers with high degree of virtualization under batch task arrivals," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2429–2438, 2012.

[116] H. Khazaei, J. Misic, and V. B. Misic, "Modelling of cloud computing centers using m/g/m queues," in *2011 31st International Conference on Distributed Computing Systems Workshops*, IEEE, 2011, pp. 87–92.

[117] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, IBM Corp., 2017, pp. 234–240.

[118] I. Kholod *et al.*, "Open-source federated learning frameworks for iot: A comparative review and analysis," *Sensors*, vol. 21, no. 1, p. 167, 2021.

[119] H. Kim, J. Park, M. Bennis, and S.-L. Kim, "Blockchained on-device federated learning," *IEEE Communications Letters*, vol. 24, no. 6, pp. 1279–1283, 2019.

[120] J. Kim, D. Kim, and J. Lee, "Design and implementation of kubernetes enabled federated learning platform," in *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, IEEE, 2021, pp. 410–412.

[121] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. arXiv: 1412.6980 [cs.LG].

[122] S. Kitajima and N. Matsuoka, "Inferring calling relationship based on external observation for microservice architecture," in *International Conference on Service-Oriented Computing*, Springer, 2017, pp. 229–237.

[123] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: building more robust cloud applications," *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pp. 700–711, 2014, ISSN: 00092614.

[124] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *2017 IEEE International Conference on Software Architecture (ICSA)*, IEEE, 2017, pp. 11–20.

[125] J. Konečnỳ, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv preprint arXiv:1610.02527*, 2016.

[126] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[127] C. Korkmaz, H. E. Kocas, A. Uysal, A. Masry, O. Ozkasap, and B. Akgun, "Chain fl: Decentralized federated machine learning via blockchain," in *2020 Second international conference on blockchain computing and applications (BCCA)*, IEEE, 2020, pp. 140–146.

[128] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[129] Kubernetes, *Production-grade container orchestration*, https://kubernetes.io.

[130] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?" *Empirical Software Engineering*, vol. 24, no. 4, pp. 2469–2508, 2019.

[131] A. Lalitha, O. C. Kilinc, T. Javidi, and F. Koushanfar, "Peer-to-peer federated learning on graphs," *arXiv preprint arXiv:1901.11173*, 2019.

[132] A. Lalitha, S. Shekhar, T. Javidi, and F. Koushanfar, "Fully decentralized federated learning," in *Third workshop on Bayesian Deep Learning (NeurIPS)*, 2018.

[133] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[134] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[135] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in Java-based open source projects," in *ACM/SPEC International Conference on Performance Engineering*, 2017, pp. 373–384.

[136] P. Leitner and J. Cito, "Patterns in the chaos—a study of performance variation and predictability in public iaas clouds," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 3, pp. 1–23, 2016.

[137] L. Li, Y. Fan, M. Tse, and K.-Y. Lin, "A review of applications in federated learning," *Computers & Industrial Engineering*, vol. 149, p. 106 854, 2020.

[138] Q. Li *et al.*, "A survey on federated learning systems: Vision, hype and reality for data privacy and protection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[139] S. Li *et al.*, "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Information and Software Technology*, vol. 131, p. 106 449, 2021.

[140] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[141] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 429–450, 2020.

[142] Z. Li, V. Sharma, and S. P. Mohanty, "Preserving data privacy via federated learning: Challenges and solutions," *IEEE Consumer Electronics Magazine*, vol. 9, no. 3, pp. 8–16, 2020.

[143] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent," *Advances in neural information processing systems*, vol. 30, 2017.

[144] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: Challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ser. ACDC '09, Barcelona, Spain: ACM, 2009, pp. 13–18, ISBN: 978-1-60558-585-7.

[145] W. Y. B. Lim *et al.*, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[146] A. Lingayat, R. R. Badre, and A. K. Gupta, "Performance evaluation for deploying Docker containers on baremetal and virtual machine," in *3rd International Conference on Communication and Electronics Systems (ICCES)*, IEEE, 2018, pp. 1019–1023.

[147] S. K. Lo, Q. Lu, C. Wang, H.-Y. Paik, and L. Zhu, "A systematic literature review on federated machine learning: From a software engineering perspective," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–39, 2021.

[148] S. K. Lo, Q. Lu, L. Zhu, H.-Y. Paik, X. Xu, and C. Wang, "Architectural patterns for the design of federated learning systems," *Journal of Systems and Software*, vol. 191, p. 111 357, 2022.

[149] Locust, *Locust - A modern load testing framework*, Last accessed 2021-10-12, 2021. [Online]. Available: http://locust.io/.

[150] J. D. Long, D. Feng, and N. Cliff, "Ordinal analysis of behavioral data," *Handbook of psychology*, pp. 635–661, 2003.

[151] M. Maggio, C. Klein, and K. E. Årzén, "Control strategies for predictable brownouts in cloud computing," *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 19, pp. 689–694, 2014, ISSN: 14746670.

[152] H. Malik, "A methodology to support load test analysis," in *ACM/IEEE 32nd International Conference on Software Engineering*, IEEE, vol. 2, 2010, pp. 421–424.

[153] S. U. Malik, S. U. Khan, and S. K. Srinivasan, "Modeling and analysis of state-of-the-art vm-based cloud management platforms," *IEEE Transactions on Cloud Computing*, vol. 1, no. 1, pp. 1–1, 2013.

[154] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50 –60, 1947. DOI: 10.1214/aoms/1177730491. [Online]. Available: https://doi.org/10.1214/aoms/1177730491.

[155] B. Mayer and R. Weinreich, "An approach to extract the architecture of microservice-based software systems," in *2018 IEEE symposium on service-oriented system engineering (SOSE)*, IEEE, 2018, pp. 21–30.

[156] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[157] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," *arXiv preprint arXiv:1710.06963*, 2017.

[158] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 691–706.

[159] P. Mell, T. Grance, *et al.*, "The nist definition of cloud computing," 2011.

[160] P. Mellor, *Optimizing kafka broker configuration*, 2021. [Online]. Available: https://strimzi.io/blog/2021/06/08/broker-tuning/.

[161] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[162] Mesosphere, Inc., *Marathon Recipes*, https://mesosphere.github.io, Dec. 2018. [Online]. Available: https://mesosphere.github.io/marathon/docs/recipes.

[163] G. Mezzetti, A. Møller, and M. T. Torp, "Type regression testing to detect breaking changes in Node.js libraries," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[164] H. Michael, *DevOps for Developers*. Apress, 2012.

[165] I. Miell and A. H. Sayers, *Docker in Practice*, 1st. USA: Manning Publications Co., 2016, ISBN: 1617292729.

[166] N. M. Mohammed, M. Niazi, M. Alshayeb, and S. Mahmood, "Exploring software security approaches in software development lifecycle: A systematic mapping study," *Computer Standards & Interfaces*, vol. 50, pp. 107–115, 2017.

[167] I. Mohiuddin, *Home page - cybera - alberta's not-for-profit digital accelerator*, 2019. [Online]. Available: https://www.cybera.ca/.

[168] N. Mostafa and C. Krintz, "Tracking performance across software revisions," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 2009, pp. 162–171.

[169] M. A. Mukwevho and T. Celik, "Toward a smart cloud: A review of fault-tolerance methods in cloud systems," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 589–605, 2018.

[170] A. Nandi, F. Xhafa, and R. Kumar, "A docker-based federated learning framework design and deployment for multi-modal data stream classification," *Computing*, pp. 1–35, 2023.

[171] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 2012, pp. 299–310.

[172] T. H. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 232–241.

[173] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge," in *ICC 2019-2019 IEEE international conference on communications (ICC)*, IEEE, 2019, pp. 1–7.

[174] T. Nylander, C. Klein, K. E. Årzén, and M. Maggio, "BrownoutCC: Cascaded Control for Bounding the Response Times of Cloud Applications," *Proceedings of the American Control Conference*, vol. 2018-June, pp. 3354–3361, 2018, ISSN: 07431619.

[175] OpenMined, *A world where every good question is answered*, https://www.openmined.org.

[176] F. Ornelas-Tellez, J. J. Rico-Melgoza, A. E. Villafuerte, and F. J. Zavala-Mendoza, *Neural Networks: A Methodology for Modeling and Control Design of Dynamical Systems*. Elsevier Inc., 2019, pp. 21–38, ISBN: 9780128182475.

[177] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[178] A. V. Papadopoulos *et al.*, "Control-based load-balancing techniques: Analysis and performance evaluation via a randomized optimization approach," *Control Engineering Practice*, vol. 52, pp. 24–34, 2016, ISSN: 09670661.

[179] J. Peng and R. Dubay, "Identification and adaptive neural network control of a DC motor system with dead-zone characteristics," *ISA Transactions*, vol. 50, no. 4, pp. 588–598, 2011, ISSN: 00190578.

[180] A. M. Potdar, D. Narayan, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020.

[181] Preston-Werner, Tom, *Semantic Versioning 2.0.0*, Last accessed 2021-02-18, Feb. 2021. [Online]. Available: https://semver.org/.

[182] Y. Qi, M. S. Hossain, J. Nie, and X. Li, "Privacy-preserving blockchain-based federated learning for traffic flow prediction," *Future Generation Computer Systems*, vol. 117, pp. 328–337, 2021.

[183] H. Qian, D. Medhi, and K. Trivedi, "A hierarchical model to evaluate quality of experience of online services hosted by cloud computing," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, IEEE, 2011, pp. 105–112.

[184] P. K. Quan, M. Kundroo, and T. Kim, "Experimental evaluation and analysis of federated learning in edge computing environments," *IEEE Access*, vol. 11, pp. 33 628–33 639, 2023.

[185] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[186] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[187] S. Reddi *et al.*, "Adaptive federated optimization," *arXiv preprint arXiv:2003.00295*, 2020.

[188] P. Regulation, "Regulation (eu) 2016/679 of the european parliament and of the council," *Regulation (eu)*, vol. 679, p. 2016, 2016.

[189] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability prediction for component-based software architectures," *Journal of systems and software*, vol. 66, no. 3, pp. 241–252, 2003.

[190] C. Richardson, *Microservices pattern: Monolithic architecture pattern*. [Online]. Available: https://microservices.io/patterns/monolithic.html.

[191] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[192] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen's d indices the most appropriate choices," in *Annual Meeting of the Southern Association for Institutional Research*, 2006.

[193] A. G. Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger, "Braintorrent: A peer-to-peer environment for decentralized federated learning," *arXiv preprint arXiv:1905.06731*, 2019.

[194] B. Ruan, H. Huang, S. Wu, and H. Jin, "A performance study of containers in cloud environment," in *Asia-Pacific Services Computing Conference*, Springer, 2016, pp. 343–356.

[195] T. Ryffel *et al.*, "A generic framework for privacy preserving deep learning," *arXiv preprint arXiv:1811.04017*, 2018.

[196] M. Sabuhi, N. Mahmoudi, and H. Khazaei, "Optimizing the performance of containerized cloud software systems using adaptive pid controllers," *ACM Trans. Auton. Adapt. Syst.*, vol. 15, no. 3, 2021, ISSN: 1556-4665. DOI: 10.1145/3465630. [Online]. Available: https://doi.org/10.1145/3465630.

[197] M. Sabuhi, P. Musilek, and C.-P. Bezemer, "Micro-fl: A fault-tolerant scalable microservice based federated learning platform," *IEEE Internet of Things Journal*, 2023, under review.

[198] M. Sabuhi, P. Musilek, and C.-P. Bezemer, "Studying the performance risks of upgrading docker hub images: A case study of wordpress," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22, Bejing, China: Association for Computing Machinery, 2022, 97–104, ISBN: 9781450391436. DOI: 10.1145/3489525.3511683. [Online]. Available: https://doi.org/10.1145/3489525.3511683.

[199] M. Sabuhi, M. Zhou, C.-P. Bezemer, and P. Musilek, "Applications of generative adversarial networks in anomaly detection: A systematic literature review," *IEEE Access*, vol. 9, pp. 161 003–161 029, 2021. DOI: 10.1109/ACCESS.2021.3131949.

[200] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," in *21st twente student conference on IT*, vol. 21, 2014, pp. 1–7.

[201] J. Scheuner and P. Leitner, "Estimating cloud application performance based on micro-benchmark profiling," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 90–97.

[202] Y. Sharma, B. Javadi, W. Si, and D. Sun, "Reliability and energy efficiency in cloud computing systems: Survey and taxonomy," *Journal of Network and Computer Applications*, vol. 74, pp. 66–85, 2016.

[203] V. Shejwalkar and A. Houmansadr, "Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning," in *NDSS*, 2021.

[204] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 82–89.

[205] Sherpa, *Privacy-preserving artificial intelligence to accelerate your business*, https://sherpa.ai.

[206] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 784–810, 2017.

[207] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker Hub," *CODASPY - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*, pp. 269–280, 2017.

[208] B. Sniezynski, P. Nawrocki, M. Wilk, M. Jarzab, and K. Zielinski, "Vm reservation plan adaptation using machine learning in cloud computing," *Journal of Grid Computing*, pp. 1–16, 2019.

[209] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, 2007, pp. 275–287.

[210] J. Steinhardt, P. W. W. Koh, and P. S. Liang, "Certified defenses for data poisoning attacks," *Advances in neural information processing systems*, vol. 30, 2017.

[211] S. U. Stich, "Local sgd converges fast and communicates little," *arXiv preprint arXiv:1805.09767*, 2018.

[212] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for microservices-based cloud applications," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2015, pp. 50–57.

[213] V. Tarasov, L. Rupprecht, D. Skourtis, W. Li, R. Rangaswami, and M. Zhao, "Evaluating docker storage performance: From workloads to graph drivers," *Cluster Computing*, vol. 22, no. 4, pp. 1159–1172, 2019.

[214] Y. Tian, Z. Guo, J. Zhang, and Z. Al-Ars, "Dfl: High-performance blockchain-based federated learning," *arXiv preprint arXiv:2110.15457*, 2021.

[215] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds, "An architecture for self-managing microservices," in *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, 2015, pp. 19–24.

[216] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, "Self-managing cloud-native applications: Design, implementation, and experience," *Future Generation Computer Systems*, vol. 72, pp. 165–179, 2017.

[217] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu, "Data poisoning attacks against federated learning systems," in *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, Springer, 2020, pp. 480–501.

[218] M. Tomisa, M. Milkovic, and M. Cacic, "Performance Evaluation of Dynamic and Static WordPress-based Websites," *ICSEC 2019 - 23rd International Computer Science and Engineering Conference*, pp. 321–324, 2019.

[219] S. Truex *et al.*, "A hybrid approach to privacy-preserving federated learning," in *Proceedings of the 12th ACM workshop on artificial intelligence and security*, 2019, pp. 1–11.

[220] A. Ullah, J. Li, Y. Shen, and A. Hussain, "A control theoretical view of cloud elasticity: taxonomy, survey and challenges," *Cluster Computing*, vol. 21, no. 4, pp. 1735–1764, 2018, ISSN: 15737543.

[221] C. Van Berkel, "Multi-core for mobile phones," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2009, pp. 1260–1265.

[222] P. Vanhaesebrouck, A. Bellet, and M. Tommasi, "Decentralized collaborative learning of personalized models over networks," in *Artificial Intelligence and Statistics*, PMLR, 2017, pp. 509–517.

[223] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[224] M. Villamizar *et al.*, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, IEEE, 2015, pp. 583–590.

[225] W3Techs, *Usage statistics and market share of WordPress*, Last accessed 2021-10-12, 2021. [Online]. Available: https://w3techs.com/technologies/details/cm-wordpress/all/all.

[226] J. Wang, A. Hertzmann, and D. J. Fleet, "Gaussian process dynamical models," *Advances in neural information processing systems*, vol. 18, 2005.

[227] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor, "Tackling the objective inconsistency problem in heterogeneous federated optimization," *Advances in neural information processing systems*, vol. 33, pp. 7611–7623, 2020.

[228] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *2008 10th ieee international conference on high performance computing and communications*, Ieee, 2008, pp. 825–830.

[229] R. Wang and W.-T. Tsai, "Asynchronous federated learning system based on permissioned blockchains," *Sensors*, vol. 22, no. 4, p. 1672, 2022.

[230] Y. Wang, J. Zhou, G. Feng, X. Niu, and S. Qin, "Blockchain assisted federated learning for enabling network edge intelligence," *IEEE Network*, 2022.

[231] Wappalyzer, *Identify technology on websites.* Last accessed 2021-10-12, Dec. 2021. [Online]. Available: https://www.wappalyzer.com/.

[232] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[233] N. Weber, P. Holzer, T. Jacob, and E. Ramentol, "Fed-dart and fact: A solution for federated learning in a production environment," *arXiv preprint arXiv:2205.11267*, 2022.

[234] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[235] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.

[236] J. P. Wong, "HyScale: Hybrid Scaling of Dockerized Microservices Architectures," 2018. DOI: 10.1016/j.trstmh.2007.04.023.

[237] WordPress, *Hosting requirements for WordPress*, Last accessed 2021-10-12. [Online]. Available: https://wordpress.org/about/requirements/.

[238] WordPress, *WordPress is open source software you can use to create a beautiful website, blog, or app.* Dec. 2021. [Online]. Available: https://wordpress.org/.

[239] X. Wu, Z. Wang, J. Zhao, Y. Zhang, and Y. Wu, "Fedbc: Blockchain-based decentralized federated learning," in *2020 IEEE international conference on artificial intelligence and computer applications (ICAICA)*, IEEE, 2020, pp. 217–221.

[240] K. Xiong and H. Perros, "Service performance and analysis in cloud computing," in *2009 Congress on Services-I*, IEEE, 2009, pp. 693–700.

[241] M. Xu, W. Tian, and R. Buyya, "A survey on load balancing algorithms for virtual machines placement in cloud computing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, e4123, 2017.

[242] P. Xu, S. Shi, and X. Chu, "Performance evaluation of deep learning tools in Docker containers," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, IEEE, 2017, pp. 395–403.

[243] R. Xu, N. Baracaldo, Y. Zhou, A. Anwar, and H. Ludwig, "Hybridalpha: An efficient approach for privacy-preserving federated learning," in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, 2019, pp. 13–23.

[244] Q. Yang, Y. Liu, Y. Cheng, Y. Kang, T. Chen, and H. Yu, "Federated learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 13, no. 3, pp. 1–207, 2019.

[245] S. Yuan, B. Cao, M. Peng, and Y. Sun, "Chainsfl: Blockchain-driven federated learning from design to realization," in *2021 IEEE Wireless Communications and Networking Conference (WCNC)*, IEEE, 2021, pp. 1–6.

[246] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated Docker containers, severity vulnerabilities, and bugs," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 491–501.

[247] K. Zhang, X. Song, C. Zhang, and S. Yu, "Challenges and future directions of secure federated learning: A survey," *Frontiers of computer science*, vol. 16, no. 5, pp. 1–8, 2022.

[248] L. Zhang, G. Ding, Q. Wu, Y. Zou, Z. Han, and J. Wang, "Byzantine attack and defense in cognitive radio networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1342–1363, 2015.

[249] M. Zhang, D. Marino, and P. Efstathopoulos, "Harbormaster: Policy enforcement for containers," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2015, pp. 355–362.

[250] Z. Zhang, Z. Gao, Y. Guo, and Y. Gong, "Scalable and low-latency federated learning with cooperative mobile edge networking," *IEEE Transactions on Mobile Computing*, 2022.

[251] N. Zhao *et al.*, "Large-scale analysis of docker images and performance implications for container storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 918–930, 2020.

[252] H. Zhu and I. Bayley, "If docker is the answer, what is the question?" In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, IEEE, 2018, pp. 152–163.

[253] W. Zhuang, X. Gan, Y. Wen, and S. Zhang, "Easyfl: A low-code federated learning platform for dummies," *IEEE Internet of Things Journal*, vol. 9, no. 15, pp. 13 740–13 754, 2022.

[254] X. Zhuang, S. Kim, M. i. Serrano, and J.-D. Choi, "Perfdiff: A framework for performance difference analysis in a virtual machine environment," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 4–13.