UMEÅ UNIVERSITY

A Comparison Between Different Frameworks Based on Application Metrics à la Argo Rollouts

Gustaf Söderlund

Gustaf Söderlund
Spring 2024
Degree Project in Computing Science and Engineering, 30 credits
Supervisor: Jakub Krzywda
Examiner: Henrik Björlund
Master of Science Programme in Computing Science and Engineering, 300 credits

Abstract

This research investigates the integration and effectiveness of two monitoring frameworks, the Four Golden Signals and the RED Method, with Argo Rollouts for automated deployments. The study aims to identify which framework integrates better with Argo Rollouts, compare their effectiveness in automating deployment procedures, and assess the impact of automated deployments on application performance. Experiments involve fault injections, such as HTTP 500 errors and delays, to evaluate the frameworks ability to detect unhealthy deployments and trigger rollbacks. Both frameworks were successfully integrated using Prometheus for metric collection and custom analysis templates for health assessment. The Four Golden Signals provided more comprehensive insights due to its additional metrics (saturation and latency), whereas the RED Method was simpler to configure and interpret. The findings highlight the importance of carefully calibrating metric thresholds to accurately identify unhealthy deployments. Future work suggests exploring Blue-Green deployments, investigating the robustness of systems under security breaches, and assessing cost savings from using Argo Rollouts over time.

Acknowledgements

I would like to thank my supervisor at Elastisys, Jakub Krzywda, for the support, invaluable feedback, and advice throughout the course of this thesis. The guidance has been crucial in helping me complete this work, and I am incredibly grateful for the patience and encouragement.

I would like to express my gratitude to my fellow master's thesis students at Elastisys. Their support and constructive feedback contributed to the quality of my work.

And thank you, Elastisys, for this opportunity to collaborate with you on this thesis.

Contents

1 Introduction

The usage of cloud computing [28] has increased significantly over the past two decades and is expected to continue growing in the future. Cloud computing grew mainly due to the reduced start-up costs for IT businesses. Instead of buying your equipment, a business could rent equipment from cloud service providers such as Google, Amazon, or Microsoft. Due to the fast growth of cloud computing, new tools and technology to enhance the developer experience have arrived.

Kubernetes [27] is one of the most important orchestration tools in the cloud today that allows for managing the deployments in the cloud. Therefore, the thesis will be carried out in collaboration with ***Elastisys***, a company that specializes in developing the next generation of cloud-native technology.

Historically, some software companies relied on distinct teams for development and operations. Developers were responsible for writing the code, while operators deployed and managed it. This separation often led to inefficiencies, as the deployment process was slowed down by the need for developers to wait on operators to identify and communicate issues. Consequently, this resulted in extended development cycles and delays in delivering new code changes and features to end users.

To address these inefficiencies, the DevOps [13] approach was developed. DevOps combines the previously separate roles of development and IT operations into a unified process. This integration seeks to eliminate the inefficiencies caused by the traditional divide which historically created significant barriers to communication and collaboration.

DevOps emphasizes continuous collaboration and improvement across all stages of the software lifecycle. In cloud computing, this approach is particularly effective as it involves automating processes and tools to efficiently manage and evolve applications.

Continuous deployment [32] is one of the practices DevOps uses to automate software release. Continuous deployment is a software development practice that automatically releases new code into production without human interaction. However, continuous deployment can easily lead to "breaking production at scale." To avoid this, automated rollouts and rollbacks can be performed. These techniques consist of setting up rules to monitor the application's health and only roll out a new version if the application stays healthy. The rollouts can be performed using different deployment strategies such as Canary Deployment and Blue-green deployment. Argo Rollouts [6] is a tool for managing automated rollouts and rollbacks. Argo Rollouts allows for setting up an analysis that will monitor specific metrics of the new deployment. If these metrics reach a certain threshold, the analysis will signal that it will perform a rollback.

Monitoring is necessary to ensure that the new deployment has been deployed correctly and stays healthy. But what metrics are the best? Is it possible to randomly select different metrics and start monitoring? Several frameworks are used by the industry today that will

guide what metrics to use for your system, but which framework is the one to use?

This thesis investigates how different frameworks and Argo Rollout impact the application's health differently and whether there is a difference in the reliability of the deployment. Then, it compares the deployment procedures of Argo Rollouts when exposed to various faults using metrics from different guideline monitoring frameworks and concludes which framework is preferable.

1.1 Aims and Objectives

The aim and objectives serve as a foundation for guiding the project direction and providing a limit to the scope of the project. The aim represents the overall purpose of the study, while the objectives will function as milestones.

1. Create an environment to experiment on.
   (a) Set up a Kubernetes cluster, preferably lightweight.
   (b) Identify applications that can be used for experimenting in Kubernetes.
   (c) Set up Argo Rollout.
2. Investigate different frameworks.
   (a) Investigate how these frameworks can be applied together with Argo Rollout.
   (b) Investigate how to do a rollout/ rollback using a threshold based on the metrics from the framework.
3. Investigate how such automated deployments could impact the performance of applications and/or availability and/or reliability.
   (a) Select evaluation criteria.
   (b) Analyze and evaluate the performance of the applications.

1.2 Thesis Outline

An outline explaining each chapter of the thesis.

The first Chapter provides a brief overview of the thesis, including an introduction to the subject, a problem formulation, and aims and objectives.

The second chapter contains a literature study and background information that is relevant to the thesis. It covers topics such as Agile methodology, CI/CD, DevOps, cloud computing, tools, deployment strategies, and monitoring frameworks.

The third chapter covers the methodology of the thesis including the experimental set up, selection of frameworks, deployment strategy, application, Kubernetes, and other tools used. The chapter explains how the experiments are conducted and how the data is collected and analyzed.

The fourth chapter presents the findings of the experiments, including visualizations of the data collected during fault injections.

The fifth chapter analyses and discusses the results from the previous chapter in the context of the research questions.

The sixth chapter reflects on technical challenges encountered and provides personal reflections on the work during the thesis. It also suggests areas of future work.

2 Literature Study

This chapter describes the background of the thesis, beginning with Agile methodology, CI/CD, and DevOps, business perspective, as Argo Rollout is a tool that is being used to enhance this area. Because the thesis is in the field of Cloud computing, an introduction to Cloud computing, virtualization, containerization, container orchestration, Kubernetes, service mesh, and Prometheus is brought up to give the reader some understanding of the field. This is followed up by a description of the different deployment strategies that are included, as well as a description of Argo Rollout and its key features. Different monitoring frameworks are described. The literature study finishes off with a discussion on related work and the research questions are introduced.

2.1 Agile Methodology

Agile development [24] is a methodology used in software development that emphasizes flexibility, continuous improvement, and a high level of customer involvement. Originating from the Agile Manifesto [2], published in 2001 by a group of software developers discussing lightweight development. Agile development seeks to address the inefficiencies and limitations of traditional software development processes like the Waterfall model [11]. The waterfall model involves sequential phases of development and has limited flexibility for changes once the project is underway. Agile development aims to break down large projects into smaller tasks that can be finished quickly. By having smaller tasks, developers can deliver value to the customer more frequently, and this can be used to evaluate the progress of the overall project.

The agile principles mentioned in the Agile Manifesto [2] aims to improve the software development process. This led to the agile principles influencing the DevOps movement, which in turn gave rise to CI/CD [10].

2.1.1 Continuous Integration and Continuous Delivery/Deployment

CI/CD [34] combines the practices of continuous integration (CI), continuous delivery, and continuous deployment (CD). CI/CD aims to automate the process for developers when new code changes have been made until the new code is in production. CI/CD allows code changes to be delivered faster and more frequently. This is done using automated test pipelines, allowing immediate feedback when the new changes fail or break.

CI is the practice of using automation to enable teams of developers to merge code changes into a shared code base. The developers need to continuously commit their code changes so that conflicts are handled quickly, this will ensure reliability when developers merge their code changes.

CD stands for both continuous delivery and continuous deployment, the purpose is the same, to deploy code changes, but is done with different strategies. Continuous delivery is the practice of delivering code changes to the production environment. Once the automated testing is finished, the code changes can then be deployed manually.

Continuous deployment is an extension of continuous delivery where it has no human intervention. This practice focuses on automation of the deployment. When the code changes has passed the automated tests of the production pipeline the code will be released to the end users with the help of automation. The main difference between continuous delivery and continuous deployment is that continuous delivery needs to be manually deployed.

2.1.2 DevOps

DevOps [13] is an approach to software development, deployment, and maintenance that combines the previously separate roles of development and IT operations into a unified process. This integration aims to eliminate inefficiencies caused by the traditional divide between developers, who write the code, and operators, who deploy and manage it. Historically, this divide had created significant communication and collaboration barriers.

DevOps emerged from the Agile movement as a response to the limitations of traditional software development methods. It extends Agile principles beyond just the development teams to operations, emphasizing continuous collaboration and improvement across all software lifecycle stages.

DevOps uses a set of practices that promote the CI/CD approach. This approach involves frequently merging code changes into a central repository, followed by automated builds and tests. The primary objective is to make the process of releasing new software updates as fast as possible while ensuring that they meet high-quality and security standards.

DevOps heavily relies on automation to speed up software changes while minimizing human involvement. This approach reduces the risk of errors and accelerates the processes.

Collaboration and Communication are crucial in DevOps. Tools and practices that support configuration management, integration, testing, and monitoring are shared across traditionally siloed teams to encourage transparency and a unified approach to problem-solving.

Continuous monitoring of the application and system performance allows teams to detect and resolve issues early, ensure optimal performance, and improve user satisfaction.

2.1.3 DevOps From a Business Perspective

The authors of the Phoenix Project novel [25] state that the capability to bring new features to market swiftly offers a substantial competitive advantage. It enhances customer satisfaction and market share, boosts employee productivity and morale, and positions organizations to succeed in a competitive marketplace. This advantage is crucial because technology has become the central mechanism for creating value and is often the primary method for acquiring customers in many organizations.

Organizations that take weeks or months to deploy software find themselves at a disadvantage. In today's fast-paced market, the ability to quickly adapt and deliver is key to maintaining competitive relevance.

The 2012 *State of DevOps Report* [25] highlights that organizations that take advantage of DevOps practices are capable of deploying code thirty times more frequently. Additionally, the duration from committing code to its deployment in production was eight thousand times faster compared to organizations not implementing DevOps methodologies. Even though this report was written twelve years ago, the 2021 *State of DevOps Report* [9] shows the same advantages. In the 2021 report they compared ”elite groups”, consisting of organizations that fully adopt DevOps practices, and ”low groups”, consisting of organizations that may not use DevOps or are inefficient in their practices. They found that ”elite groups” could deploy code 973 times more frequently, and the duration from committing code to its deployment was 6750 times faster than the ”low groups” were.

2.2 Cloud Computing

Cloud computing [28] is the area that the thesis will focus on, which is why it is important to have some background on cloud computing technology and how it is defined. The National Institute of Standards and Technology (NIST) defines cloud computing: *”Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”*.

There is a few essential characteristics of cloud computing that explain the cloud model. *On-demand self-service* allows consumers to autonomously provision computing resources like server time and network storage without human intervention. *Broad network access* ensures accessibility through standard mechanisms on diverse client platforms. *Resource pooling* utilizes a multi-tenant model, dynamically assigning computing resources based on consumer demand, with a degree of location independence. *Rapid elasticity* enables flexible scaling of capabilities to meet demand, appearing virtually unlimited to consumers. *Measured service* involves automatic resource control and optimization through metering, offering transparency to both providers and consumers.

There exists different deployment models for the cloud infrastructure. *Private cloud* is the deployment model where the cloud infrastructure is dedicated to a single organization. It may be owned, managed and operated by the organization, a third party or a combination. *Public cloud* is the deployment model that allow the cloud infrastructure to be open for the public. Organizations and businesses can now instead of paying for their own physical hardware utilize the pay-per-use concept where users only pay for the resources that they actually use [14]. These resources are being hosted at a remote data center. These data centers are managed by a cloud service provider which charges the user for a subscription based fee which in return lets the user use their computing resources [8]. Some of the most common cloud service providers, also referred as the Hyperscalers are Google cloud, Amazon web services(AWS) and Microsoft Azure. *Hybrid cloud* is the third deployment model for cloud infrastructure, it is a combination of private and public cloud. For example could the storage of data be hosted on a private cloud for security reasons while the computing are hosted by a public cloud.

From the cloud computing model defined by NIST in the article [28] the *cloud services* that the public cloud providers offer are these three ”as a service” models: *Infrastructure*

as a service (IaaS), *Platform as a Service* (PaaS) and *Software as a service* (SaaS). The three different models gives the customer and provider different responsibilities [20]. In Figure 1 we can see the different layers of resources that you and the cloud service provider manages depending on solution. Providing all the necessary resources such as hardware and software for your business yourself is called *On-premise*. Instead of investing a lot of money on hardware and software and then managing everything, cloud service providers offer different services and solutions based on how much responsibility a user is prepared to manage.

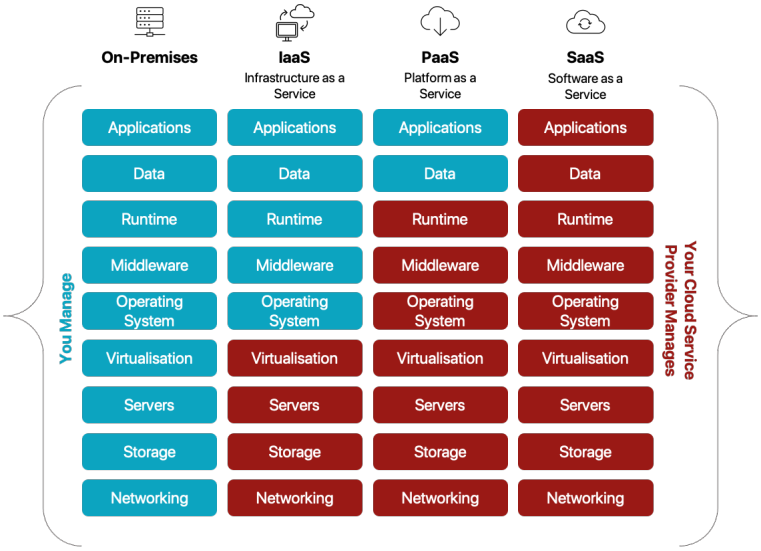

Figure 1: Cloud service models [16]

2.2.1 Virtualization

Virtualization technology [18] is the foundation of cloud computing, it is the process that allows running multiple parallel virtual environments on a single piece of infrastructure. What makes virtualization feasible is a hypervisor, which is software that can use parts of the resources of a computer, such as CPU and memory. These resources are then allocated to a virtual machine(VM) that can run its own operating system (OS). The OS that runs inside the VM is often called guest OS, where the OS that's being run in the hardware is called the host OS.

There exist two types of hypervisors, type 1 (bare-metal) and type 2 (Hosted) [18]. Bare-metal hypervisors interact directly with the hardware layer and is most common in virtual server scenarios. Hosted hypervisors interact with a OS layer and run as an application, it is more common on daily used computer when a user need to run a alternative operating system.

The two types and their architecture are presented in Figure 2. The hypervisor prevents interference between VMs, ensuring they do not intrude upon each other's memory space or compute cycles. Once the VMs are up and running the user has the possibility to execute their applications in an isolated virtual environment.

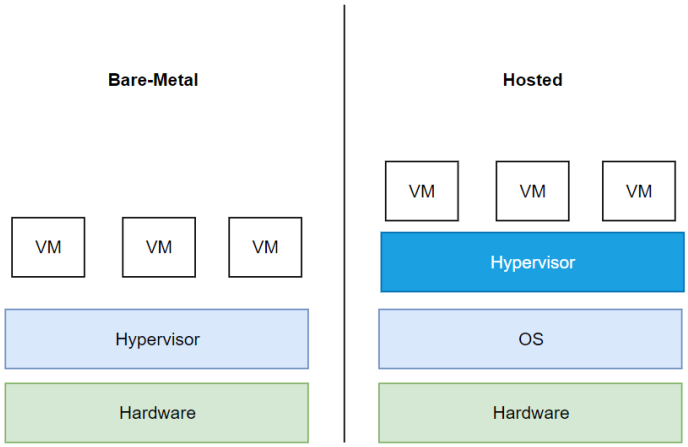

Figure 2: Architecture of virtualization using hypervisor type 1 and type 2.

2.2.2 Containerization

Containerization [17] is the packaging process of application code with all necessary files and libraries it needs to run. This creates a single lightweight executable that we call a container. Containerization allows for portability where developers can use containers in multiple environments without rewriting the application code or installing the correct version that matches the machine's OS.

Containers are a more lightweight virtualization concept than VMs, which has been the foundational element within the infrastructure layer. Containers are suggested as a more interoperable solution for application packaging in the cloud due to being less resource and time consuming [31].

Due to containers being lightweight they have high scalability because it does not require to boot an OS. This makes containers faster to launch than a traditional VM and require less capacity. Therefor several containers can be launched at the same resource capacity. Making containers more scalable and more resource efficient than VMs [17]. A visual representation is available in Figure 3 of a containerized application versus a traditionally application using VMs. The guest OS not being part of the containerized applications but being present on the virtual machines.

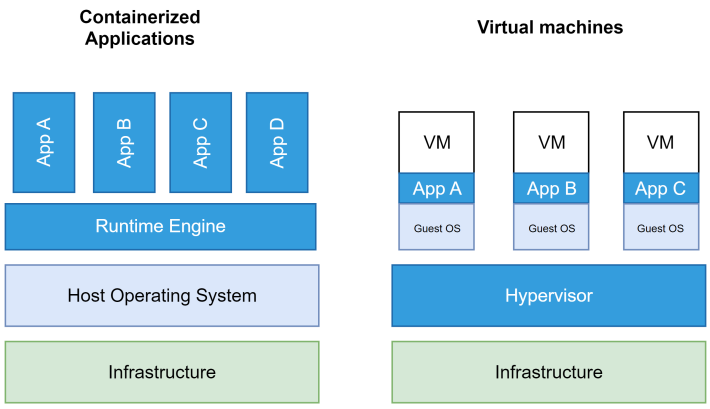

Figure 3: Architecture of Containerization compared to VM architecture

What makes the containerization possible is the runtime engine, installed on the host OS and creates the containers the runtime engine becomes a conduit for containers to share an operating system with other containers on hte same computing system [17]. An example of a runtime engine is Docker which has become very popular.

2.2.3 Container Orchestration

In environments where containers are deployed at scale, managing them individually becomes impractical. Container orchestration [35] is the automated configuration, management, and coordination of systems, applications, and services. Container orchestration helps efficiently handle the container lifecycle, including deployment, scaling, networking, and availability. By automating these processes, container orchestration tools significantly reduce the manual efforts required for deploying and scaling containerized applications.

When it comes to container orchestration tools, Kubernetes [27] has become the standard choice due to its robust feature set and active community support, although other tools such as Docker Swarm [12] and Apache Mesos [3] are also available.

2.2.4 Kubernetes

Kubernetes [27], often abbreviated as ”K8s,” is an open-source container orchestration system that revolutionized the management of containerized applications across multiple hosts. It was initially developed and open-sourced by Google in 2014 and later donated to the Cloud Native Computing Foundation in 2016. Kubernetes allows for deployment, maintenance, and scaling of applications based on various metrics such as CPU usage, memory consumption, or other predefined custom metrics.

The core of Kubernetes [26] is designed around a set of fundamental objects that define and represent the desired state of a cluster. These objects are crucial for managing containerized applications, ensuring not only scalability but also simplifying deployment processes. Among these core objects are *Pods*, which are the smallest and simplest units deployable by Kubernetes. A Pod typically represents a single instance of a running process in the cluster and can contain one or more containers that share storage and networking resources.

A *Service* enables network access to a set of Pods, providing a stable, consistent address for accessing the running containers, regardless of the underlying Pod changes. *ReplicaSet* is an object that ensures a specified number of replica Pods always run at any given time. It's primarily used to maintain the stability and availability of a set of Pods.

Deployment provides declarative updates for Pods and ReplicaSets, allowing for the application's Pods to describe the desired state.

A *Namespace* provides a way to divide the resources in the cluster and ensuring that the resources in one Namespace does not interfere with another.

2.2.5 Service Mesh

A service mesh [36] is an infrastructure layer dedicated to handling communication between services. It's designed to manage a large volume of network-based interprocess communication among services in a transparent and reliable manner. The primary objective of a service mesh is to offer a means of controlling how different components of an application share data with one another.

Istio [22] is one popular service mesh. It extends the basic functionality provided by Kubernetes to manage components of an application more efficiently and securely.

2.2.6 Prometheus

Prometheus [33] is a popular open-source monitoring and alerting toolkit used in cloud computing. It was first developed at SoundCloud in 2012 and later adopted by the Cloud Native Computing Foundation (CNCF).

Prometheus collects time series data based on metric names and key/value pairs, so called labels. This allows for accurate data collection and querying.

Prometheus Query Language (PromQL) is a unique and flexible query language designed specifically for Prometheus. It enables users to extract and manipulate data in a precise and detailed manner, making it particularly useful for monitoring.

The metrics that Prometheus scrapes can be displayed on Grafana dashboards or stored in CSV files for future use.

2.3 Deployment Strategies

When code changes has been made and a new version is ready to be deployed, the Pods with containers running the old version needs to be removed and new pods with containers running the new version needs to be deployed. This will either lead to some downtime when none of the versions are active. Another way is to increase the resources required, have both versions running concurrently, and then remove the old version, such as in a blue-green deployment. If the update to the new version is performed manually, it could lead to human errors and create a bottleneck. Kubernetes uses something called a deployment which allows for describing how the applications should be updated using different deployment strategies [19].

The process of releasing new versions of an application gradually and monitoring the new deployment until it is fully scaled up is called progressive delivery [41] such as canary-deployment and blue-green. Kubernetes does not provide a tool to analyze if the new deployment was correct or not. This is where Argo Rollouts will enhance the system and implement progressive delivery.

2.3.1 Recreate

A recreate deployment [4] is fairly straight forward, it deletes the pods with the old version and then deploys pods with the new version. This strategy does not allow for zero downtime due to the versions never run concurrent, it could be minutes, hours or even days until the new version is up and running. But if concurrency is not sought after this is a effective strategy [40]. In Figure 4, a visual representation of a recreate deployment is shown, the time between the instances of available pods running containers of the old version, blue line, and the instances of available pods running containers with the new version, green dashed line, there is a gap which represents the downtime during the two versions of the application.

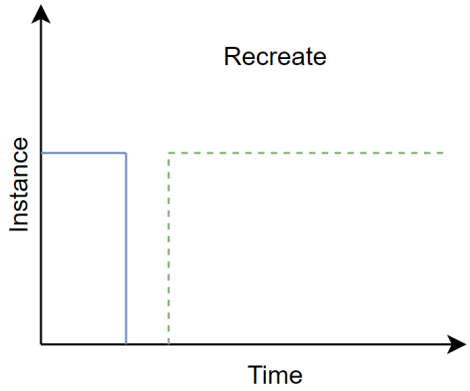

Figure 4: A recreate deployment is being performed for the number of instances during a period of time. Downtime of the application occurs.

2.3.2 Rolling Update

Rolling update [4] is the default deployment strategy in kubernetes. It allows for zero down time and the way it works is that it roll out a new version of an application iterative. When the new version of the application is being scaled up, the old application is being scaled down [40]. In Figure 5 a visual representation of a rolling update is shown. Here the number of instances of available pods running containers with the old version decreases with the blue line, as the number of instances of available pods running containers with the new version increases with the green dashed line.

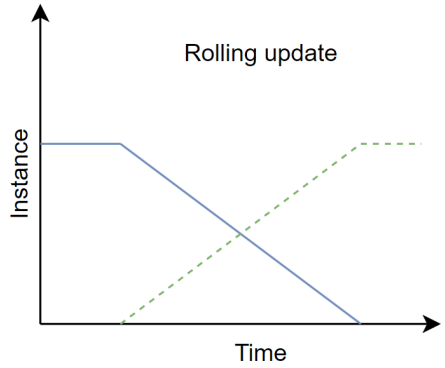

Figure 5: A rolling update is being performed for the number of instances during a period of time

2.3.3 Blue-Green

The blue-green deployment [4] (also referred to as red-black) minimizes downtime and risk. A deployment with the new version of the containers is deployed (green) concurrent with the old version of containers (Blue). The new containers (green) will not receive any traffic until they are considered healthy, once the new containers (green) is ready the traffic is switched and the number of containers (blue) of the old version is removed. The downside with this strategy is that when the new green version is deployed and until the old blue version is removed the overall system will require twice the resource capacity to run both the blue and green versions.

In Figure 6 a visual representation of the deployment is being performed. Once the new, green, version is deployed concurrent with the old, blue, version double the resource capacity is needed. When the new version is ready and available to receive traffic they switch and the old version gets removed.

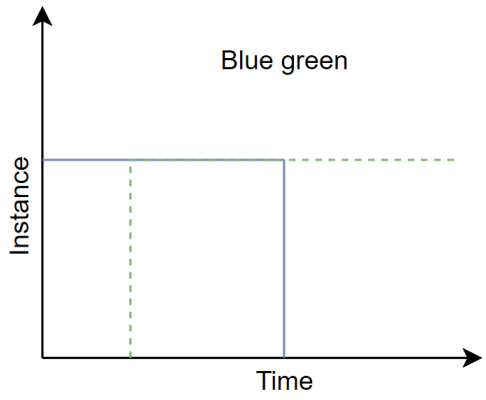

Figure 6: A blue green deployment where the blue line represent the number of instances of the old version of the application and the green dashed line represents the new version.

2.3.4 Canary Deployment

A canary deployment [4] is a technique used for deploying new versions of an application in a controlled manner. In this approach, a small subset of users are exposed to the new version, while the majority of users still use the old version. For example, only 10% of users are exposed to the new version, while 90% continue to use the old version.

This method enables one to test the new version in a real-world environment with limited exposure to users. If the new version runs smoothly, with no signs of failure, or if a specific amount of time has passed, the canary deployment proceeds to the next stage. In the next stage, more users are exposed to the new version of the application.

Figure 7 gives a visual representation of a canary deployment where the availability of the new pods running containers with the new version increases with the green dashed line when exposed to a small subset of users. The blue line represents the availability of the pods running the old version, which decreases at the rate as the new version increases.

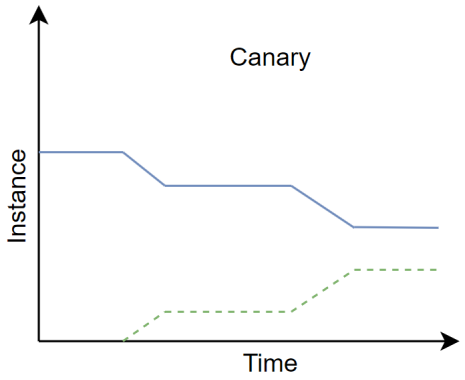

Figure 7: A Canary deployment. Blue line represents the old version of the application, the green dashed line represents the new version of the application.

2.4 Argo Rollout

Argo Rollouts [6] is a Kubernetes tool that allow for progressive delivery, enabling advanced deployment options like Blue-Green [4] and Canary deployment [4]. The primary focus is ensuring safer and controlled rollouts of new features and updates.

The tool uses Custom Kubernetes Resource Definitions (CRD), more specifically, the Rollout CRD, to manage the rollout lifecycle. The Rollout Controller manages updates based on the defined strategies.

Argo Rollouts provides deployment techniques that offer robust mechanisms for handling software updates. For example, Blue-Green Deployment reduces risks by keeping two production environments active simultaneously. This allows for instant rollback if any issues arise during deployment, ensuring a smooth transition with minimal downtime. Canary Deployment allows for a phased rollout to a limited audience before full deployment, which is ideal for catching unforeseen issues early on and not exposing potential issues to a larger audience. When performing a canary deployment, the Traffic Routing can be weighted,

which enables granular control over traffic distribution between old and new versions based on predefined criteria. Another technique used to minimize downtime is Automated Rollbacks, which automatically revert to the previous stable version in case of a deployment failure.

Another key feature of Argo Rollouts is the AnalysisTemplate resource, which evaluates the health of a new version during a Canary or Blue-Green deployment.

The Analysis Template defines the criteria for assessing the health of the new version during a rollout. It specifies the metrics to be collected and the conditions under which the rollout should succeed or fail. The template usually includes queries to monitoring systems and a definition of success criteria based on the response.

During a rollout, Argo Rollouts evaluates the specified metrics at defined intervals. If the metrics meet the success criteria, the rollout continues. If they don't, the rollout may halt or rollback, depending on the configuration.

Argo Rollouts integrates with multiple metrics providers to enable real-time decision-making during deployment processes. This integration is fundamental in implementing advanced deployment strategies such as Canary releases and Blue-Green deployments, Argo Rollouts can obtain and analyze application-specific metrics to determine the health and viability of a new deployment. One of such metric providers is Prometheus [33].

2.5 Monitoring Framework

Effective monitoring is essential to maintain the health, performance, and availability of an application. Three noteworthy frameworks for achieving this are *The Four Golden Signals*, *RED Method*, and the *Use Method*. These frameworks utilize methodologies to provide strong system performance, identify anomalies, and facilitate timely interventions. Consequently, they play a vital role in establishing a dependable and efficient operational environment.

2.5.1 USE-Method

The USE-Method [15] consists of a set of principles designed for monitoring and analyzing system performance, the acronym USE representing Utilization, Saturation, and Errors. Developed by Brendan Gregg, this method is centered around understanding and enhancing the utilization of resources within a system.

Utilization measures how much a resource (CPU, disk, network, etc.) is used. It represents the average time that the resource was busy servicing work and can indicate if the resource is a bottleneck.

Saturation refers to the extent to which a resource has excess work that it cannot handle, often resulting in a queue. Saturation is an important signal that tells if a resource is becoming a bottleneck, and it can predict performance degradation.

Errors indicate critical hardware or software issues. Monitoring them helps to detect problems early in order to prevent any impact on the system.

2.5.2 RED Method

The RED Method [1] is a monitoring framework designed to give insights into the performance and health of microservices within a software system. It focuses on three key metrics: Rate, Errors, and Duration. This method is particularly useful in distributed systems and service-oriented architectures, such as those commonly found in cloud-native environments.

Rate tracks the number of requests that a service processes over a given time period. This metric helps identify the demand being placed on a service and can indicate when a service is experiencing higher or lower traffic than normal.

Error refers to the number of failed requests within a system. This could include any request that has not been successful due to server errors, timeouts, or the inability to fulfill the request. It is important to keep track of error rates to ensure the reliability and quality of the service.

Duration is the measure of time required for a service to process a request. This metric is critical to evaluate the performance of the service. Long duration times may indicate bottlenecks or performance issues within the service, which may negatively impact the user experience.

2.5.3 The Four Golden Signals

The Four Golden Signals [7] is a set of key performance indicators that are particularly useful for monitoring and managing the reliability and performance of systems. These signals were introduced by Google as a practical approach to identifying and addressing issues in distributed systems. These signals include:

Latency refers to the time that it takes to process a request. High latency can have a negative impact on the user experience, and it may also indicate underlying issues with the service or network.

Traffic refers to the level of demand placed on your system, which can be measured by factors such as the number of requests per second, the amount of data being accessed, or the number of users accessing the system concurrently. Monitoring traffic makes it easier to understand the workload impact on the system and determine when scaling may be necessary.

Error is the rate of request failures. This could be measured in the rate of HTTP error codes returned, exceptions thrown, or other service-specific metrics of failures. High error rates can signal problems in the application or dependencies that need immediate attention.

Saturation is a measure of the capacity of the system. Saturation can indicate how close to reaching a resource's limits, which can help predict and prevent potential overloads before they occur. Measures the degree to which a system resource (e.g., CPU, memory, disk, network) is utilized.

2.6 Discussion

This literature study provides a comprehensive overview of the fundamental theories and practices relevant to the thesis, focusing primarily on Agile methodologies, Cloud Computing, and various deployment strategies as they relate to Argo Rollouts, on which the experiment will be conducted.

There exists some related work that provides parallels to using Argo Rollouts for controlled software deployment:

In the research paper *Comparison of zero downtime based deployment techniques in public cloud infrastructure* [37] there is an investigation on different deployment strategies (e.g., Blue-Green, Canary, and Rolling update). The paper doesn't mention anything special about Argo Rollouts or different monitoring frameworks. But it can provide information how these strategies are implemented in cloud environments.

Another research paper *An Evaluation of Canary Deployment Tools* [38] provides insightful information on how Argo Rollouts performance compares to other canary deployment tools such as Flagger and Spinnaker.

In the research paper *A Comparison of CI/CD tools on Kubernetes* [23] a comparison of different CI/CD tools such as Drone, Argo Workflows, ArgoCD, and GoCD are compared, even though they are not directly related to Argo Rollouts, it is a tool that improves the process of deploying applications in CI/CD systems. The paper mentions future work that focuses on handling flaws that might occur in the application and if there is a way to go back to the previous versions of the application to ensure high availability. This is the primary objective of the Argo Rollouts tool, and it is worth exploring its reliability to determine how effective it is.

Three research questions have emerged based on the literature study and background.

RQ1 Which framework can be integrated with Argo Rollouts?

RQ2 How do these frameworks compare to each other, and how can they be used to automate the deployment procedure?

RQ3 How such automated deployments could impact the performance of applications?

3 Methodology

This chapter explains the methodology used to conduct experiments aimed at answering the research questions. First, the tools, deployment strategy, environment, and application that are being used for the experiments are presented. After that, a description of how the experiments will be performed and how data will be collected is provided.

3.1 Setup

This section describes the tools, deployment strategy, environment, and application used for the experiments.

3.1.1 Selection of Frameworks

To answer the research question **RQ1** two frameworks will be used for comparison. The frameworks that are being analyzed are Four Golden Signals [7] and the RED Method [1]. Both frameworks are similar in some aspects. The *Four Golden Signals* measures latency, saturation, traffic, and error rate, while the *RED Method* measures the rate of traffic, errors, and duration. The two frameworks have some similarities and differences, so a comparison can be made to identify how each framework affects the deployment procedure.

The reason for excluding the USE-method [15] (Utilization, Saturation, Errors) is that this framework primarily focuses on the utilization and saturation of resources, such as CPU, memory, and network, which are good for diagnosing hardware issues and resource bottlenecks. However, this framework lacks direct metrics for evaluating the performance and reliability of services at the application level, such as request rates, error rates, and response times. The USE method is better suited for infrastructure monitoring rather than application performance monitoring. Thus, it was determined that the Four Golden Signals and the RED Method offer a more relevant and effective set of metrics for the objectives of this study.

3.1.2 Selection of Deployment Strategy

The Canary deployment strategy [4] was chosen as the main focus due to its incremental and risk-averse nature. This approach enables the gradual exposure of a subset of users to new software versions before a full rollout, allowing for a controlled rollout. Implementing Canary deployments enables immediate feedback on system performance from the subset of affected canaries. This feedback is valuable before broader exposure. Canary deployments are more resource-efficient. It allows for the gradual introduction of changes without the need for doubling the infrastructure, thus conserving resources.

3.1.3 Application

An application is required to perform a rollout using Argo Rollouts. Rather than building an application from scratch, one can opt for an existing one that is already compatible with Kubernetes. The Argo project [5] offers a sample application that has been utilized in several demonstrations of Argo Rollout and is an ideal choice for a sample application. The application offers built-in fault injection, which can be automated by modifying the code.

3.1.4 Kubernetes Cluster

A single-node cluster is enough to perform the experiments. For this project, MicroK8s [30] will be the tool chosen to generate a single-node cluster. MicroK8s is designed to be lightweight, making it easy to manage smaller Kubernetes clusters. MicroK8s also comes with several add-ons, which allow for easily adding tools to the cluster. The installation of MicroK8s was done using MicroK8s own *Get started* [29] documentation

The Kubernetes cluster will be run on a Virtual Machine hosted on Google Cloud Platform [14], and the following specifications were used for the Virtual Machine:

- Machine type: e2-custom-4-8192, 4 vCPU and 8 GB ram
- CPU platform: Intel Broadwell
- Architecture: x86/64
- Source image: debian-12-bookworm-v20240213
- Storage: 20 GB

3.1.5 Service Mesh

A service mesh like Istio provides a solution for handling traffic management. Istio allows for precise control over the traffic sent to the stable and canary versions of the application during the canary deployment. By using Istio, Argo Rollouts can dynamically adjust the percentage of traffic that each version receives. The installation of Istio was done using the following guide [21].

3.1.6 Monitoring

During the experiments, Prometheus enhances the monitoring capabilities within the Kubernetes environment. Prometheus collects, stores, and manages data, which is used for analyzing application performance during deployment.

When used together with Argo Rollouts, Prometheus plays a huge role by integrating with analysis templates. This integration allows for the specific metrics required for the analysis to be obtained. By using Prometheus queries within Argo Rollouts analysis templates, automation of the evaluation process during deployments can me made.

When used in a Kubernetes environment with Istio, Prometheus can access the metrics provided by Istio. By utilizing PromQL, Prometheus can directly query Istio's metrics, allowing for a better understanding of traffic flow, error rates, and latency at the service mesh level.

3.2 Solution Architecture

A solution architecture has been developed to address the research questions mentioned in section 2.6. In Figure 8, the solution architecture is present, which demonstrates the Kubernetes environment when a new version of the application is being deployed. Firstly, a new canary version of the application will be deployed using Argo Rollouts. The canary deployment strategy will route a certain percentage of incoming traffic to the canary replica set using Istio. In the solution architecture, 10% of the traffic is routed to the canary. Prometheus will continuously collect metrics from the canary pod, and the AnalysisTemplate will determine whether the canary pod is healthy. If the AnalysisTemplate determines that the canary is unhealthy, a rollback to the previous stable version of the application will be performed. On the other hand, if the analysis template determines that the canary is healthy, Argo Rollouts will gradually increase the incoming traffic to the canary replica set while simultaneously decreasing the incoming traffic to the stable replica set until all the traffic is redirected to the canary version.

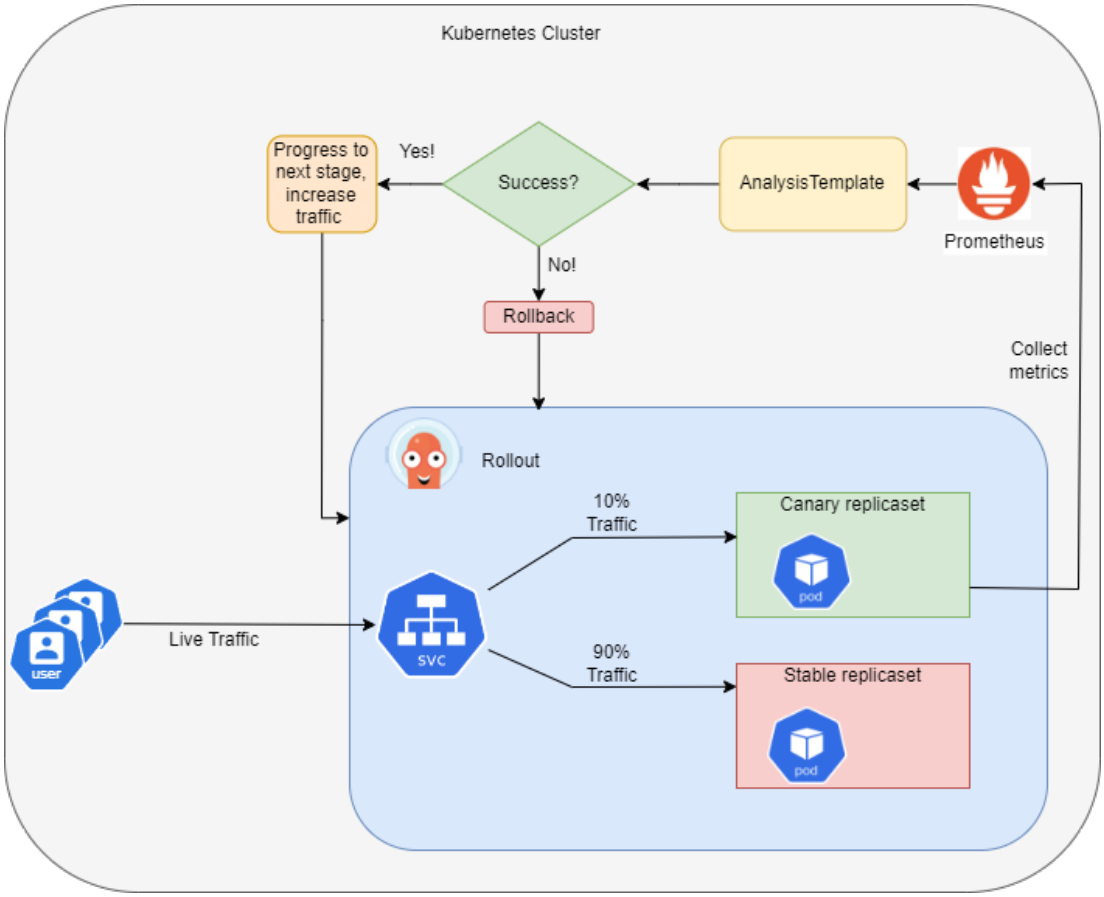

Figure 8: Solution Architecture

3.2.1 Argo Rollouts

The experiments mainly focus on the Argo Rollouts tool. Argo Projects has provided a simple guide to installing Argo Rollouts in the cluster. The commands for installing Argo Rollouts in the cluster are shown below.

```
$ microk8s kubectl create namespace argo-rollouts $ microk8s kubectl apply -n argo-rollouts
 -f https://github.com/argoproj/argo-rollouts/releases/latest/download/install.yaml
```

Once Argo Rollouts is installed in the Kubernetes environment, YAML files are needed. The YAML files needed for the experiments are present in the following git repository [39] and briefly described below.

Namespace.yaml

Namespace.yaml essentially creates a Namespace called ”rollouts-demo-istio” with the label ”istio-injection” set to ”enabled”, which indicates that Istio’s automatic sidecar injection feature is enabled for resources deployed within the Kubernetes namespace. This namespace allows for resources that require Istio service mesh capabilities.

Service.yaml

Service.yaml defines a Kubernetes Service named ”istio-rollout”. It exposes a port and forwards traffic to pods with the label ”app: istio-rollout”. It is of type NodePort, making it accessible externally on a defined port.

VirtualService.yaml

This VirtualService configures Istio to route incoming HTTP traffic for the specified hosts to either the ”stable” or ”canary” subsets of the ”istio-rollout” service based on defined weights, allowing for controlled deployments when doing canary deployments.

DestinationRule.yaml

The DestinationRule is designed to separate traffic for the ”istio-rollout” service into two subsets: ”canary” and ”stable”. Each subset is associated with pods labeled with ”app: istio-rollout”. These subsets can then be used for traffic routing during a canary deployment, where traffic is gradually shifted from one subset, ”stable”, to another, ”canary”.

Gateway.yaml

The gateway configuration defines an Istio Gateway named ”istio-rollout-gateway” that directs incoming HTTP traffic on a defined port to the pods labeled as ingress gateways within the Istio system.

Rollout.yaml

This YAML configuration sets up a rollout strategy for the ”istio-rollout” application within Argo Rollouts. It specifies a limit of two revisions for the rollout history and selects Pods based on the ”app: istio-rollout” label. It defines a container running the ”argoproj/rollouts-demo:blue” image, and sets resource limits for memory and CPU. For deployment strategy,

it employs a canary deployment approach with gradual increments and pauses between steps, in this deployment each step increases the traffic by teen percentage units every 30 second. Metadata labels distinguish stable and canary Pods. Analysis is conducted during the rollout process to ensure the stability of the canary release. Additionally, Istio is leveraged for traffic management, configuring VirtualService and DestinationRule resources to handle traffic routing.

Analystemplateredmethod.yaml

This YAML file is designed to monitor the rollout using the metrics from *RED Method* framework. Each AnalysisTemplate specifies the arguments needed (service and namespace) and the metrics to monitor, such as rate, success rate, and duration. It also specifies the conditions for success or failure, and the Prometheus query to retrieve the metric data. The addresses provided for Prometheus servers are also specified.

Analysistemplategoldensignal.yaml

This YAML file is designed to monitor the rollout using the metrics provided from *The Four Golden Signals* framework. Each AnalysisTemplate specifies the arguments needed (service and namespace), the metrics to monitor such as traffic, success rate, latency and saturation. It also specifies the conditions for success or failure, and the Prometheus query to retrieve the metric data. The addresses provided for Prometheus servers are also specified.

Kuustomize.yaml

The Kustomization YAML file utilizes Kustomize to orchestrate the deployment and configuration of various Kubernetes resources within the ”rollouts-demo-istio” namespace. It includes YAML for essential components such as namespaces, analysis templates, gateways, services, VirtualServices, rollouts, and DestinationRules. Specifying the namespace as ”rollouts-demo-istio”, the file ensures that all resources are deployed and managed within this namespace.

3.3 Experiments

This section explains the experiments conducted to compare the application performance during canary deployments using Argo Rollouts with the selected frameworks. How the thresholds for each metric is also described.

3.3.1 HTTP 500 Injection

An experiment will be conducted where HTTP 500 errors will be injected. Modifications have been made to the application’s source code, and a new docker image has been created. Once the new image is deployed to the cluster, it will begin sending HTTP 500 errors. The error rate will increase by five percentage units every thirty seconds during the experiment.

3.3.2 Delay Injection

The application executes a delay injection for the second experiment. To achieve this, a new docker image of the application was created, which triggers a sleep function. The sleep function starts with zero seconds and gradually increases by one second every minute.

3.3.3 Delay and HTTP 500 Injection

For the third experiment, a combination of the previous two experiments was conducted. A third docker image of the application was created, which contained modified code to introduce delays and HTTP 500 errors. The frequency of errors would increase by five percentage units every thirty seconds, and the delay would increase by one second every thirty seconds.

3.3.4 Establishing Baseline Thresholds

To determine an appropriate threshold for system failure during canary deployments, a series of seven rollouts were conducted without any fault injection. During these initial rollouts, key performance metrics provided by each framework were monitored and recorded using Prometheus. The data collected from these rollouts were stored in CSV format. A baseline threshold could be established by aggregating the data from all seven rollouts and calculating the average. This threshold serves as a critical reference point in the analysis templates for identifying system failure during the subsequent experimental rollouts. The baseline thresholds are present in the Figures below.

Figure 9 shows the average number of requests per second from seven separate canary deployments of the Argo Proj application where no type of fault injection is present. In the graph the requests received for the canary pod is present. There is a large increase in requests at the start because the canary pod starts receiving requests. The same can be said for the end of the graph at around 250 seconds, where the canary deployment is completed, and then the canary switches to the stable version. The peak traffic, which will be used as a guideline for the threshold in the analysis template, can be found between 150 and 200 seconds, where the number of requests reaches 28 requests per second. The analysis template for the Four Golden Signals will use a threshold of 30 requests per second for the traffic metric success condition, providing a safety margin for potential spikes and unexpected traffic increases. A rollback should be performed if the application exceeds 30 requests per second.

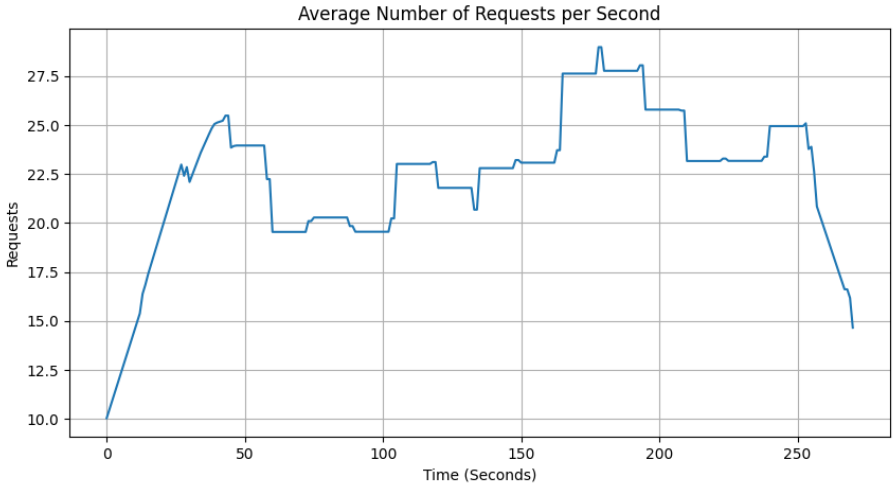

Figure 9: No fault injection, Traffic, Requests per second (The Four Golden signals). The X-axis represents the time in seconds duration of the rollout, while the Y-axis represents the number of requests.

Figure 10 shows the average success rate per second from seven separate canary deployments of the Argo Proj application where no fault injection is present. The graph shows the proportion of requests that did not return HTTP 500. Due to no HTTP 500 errors being injected, the success rate was 1.0 (100%). The analysis template for the Four Golden Signals will use a threshold of 0.9 for the Error metric as the success condition. When HTTP faults are injected, the success rate is expected to decrease, and if the success rate goes below 0.9, a rollback will be performed.

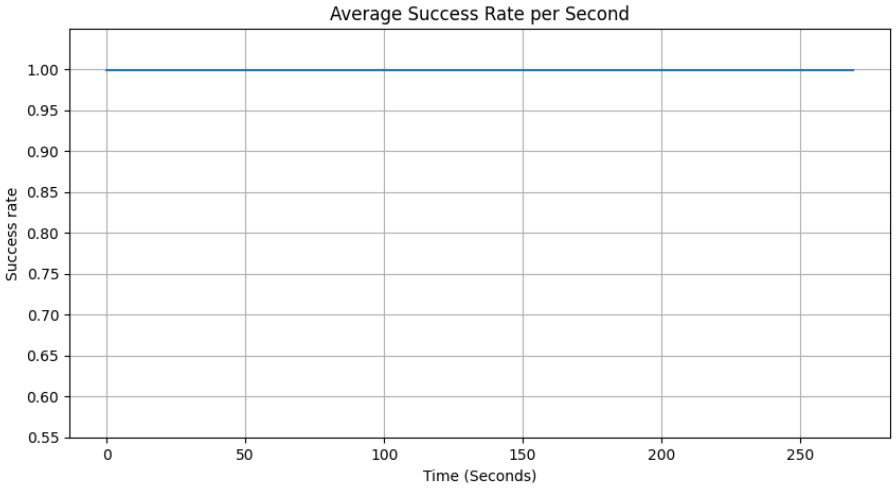

Figure 10: No fault injection, Success Rate (The Four Golden Signals). The X-axis represents the time in seconds during the rollouts, and the Y-axis represents the success rate of requests not having HTTP 500.

Figure 11 shows the average latency from seven separate canary deployments of the Argo Proj application where no fault injection is present. The graph shows a low latency, little to none, due to no delay injection. Even though the latency in the graph decreases, it is very minimal. When delay injection is present, the latency is expected to increase. Thus, the analysis template for the Four Golden Signals will use a threshold of three seconds for the latency metric during the rollouts.

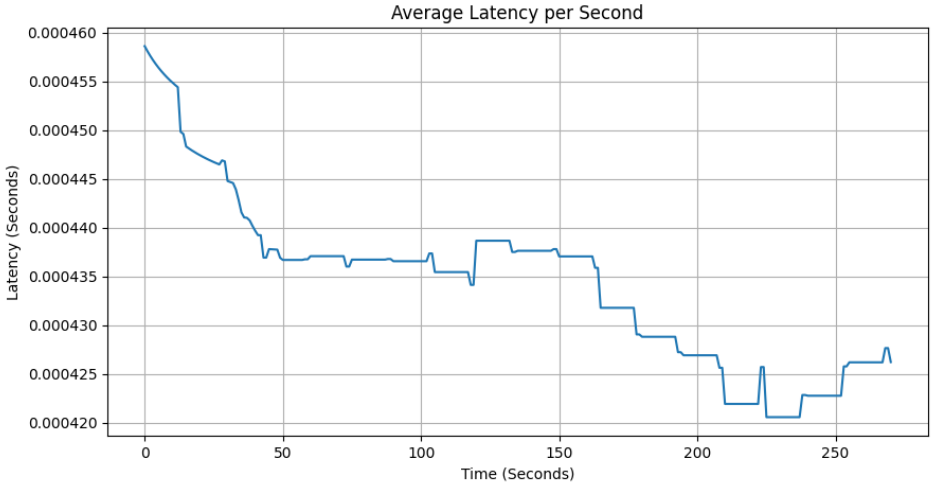

Figure 11: No fault injection, Latency (The Four Golden Signals). The X-axis represents the time in seconds during the rollouts, and The Y-axis represents the latency of requests in seconds.

Figure 12 shows the average CPU usage from seven separate canary deployments of the Argo Proj application where no fault injection is present. The average CPU usage per second of the canary pod from the seven different canary deployments reaches a maximum value that is slightly above 0,011. Thus, the threshold for saturation in the analysis template for the Four Golden Signals was set to 0,012, providing a safety margin for potential spikes and unexpected CPU usage. A rollback should be performed if the application exceeds 0,012 CPU usage per second.

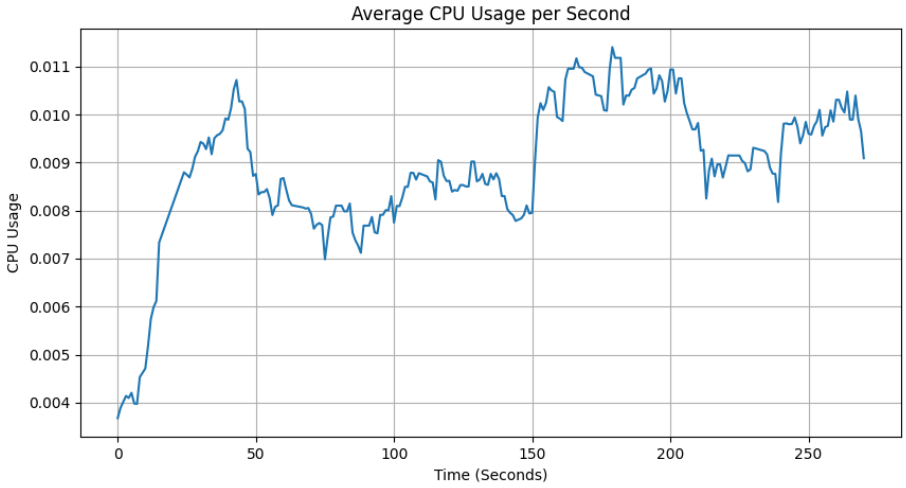

Figure 12: No fault injection, Saturation (The Four Golden Signals). The X-axis represents the time in seconds during the rollouts, and the Y-axis represents the CPU usage of the canary pods.

Figure 13 shows the average rate of requests per second from seven separate canary deployments of the Argo Proj application where no type of fault injection is present. The maximum number of requests during these seven canary deployments is roughly 28 requests per second. Thus, the threshold for the rate metric for the RED Method will use 30 requests per second in the analysis templates. Providing a safety margin for potential spikes and unexpected rate of request increases. A rollback should be performed if the application exceeds 30 requests per second.

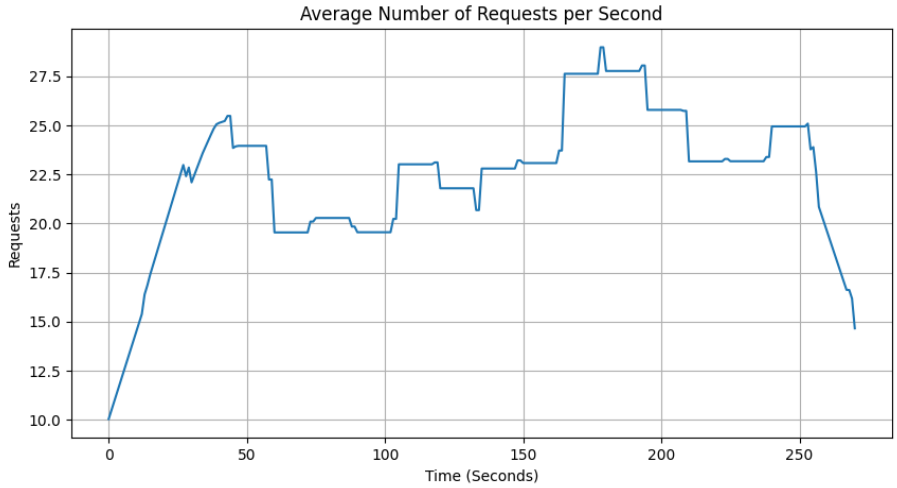

Figure 13: No fault injection, Rate of requests (RED Method). The X-axis represents the time in seconds during the rollout, while the Y-axis represents the number of requests.

Figure 14 displays the average success rate per second from seven separate canary deploy-

ments of the Argo Proj application, where no fault injection is present. The graph shows the proportion of requests that did not return HTTP 500 errors, resulting in a success rate of 1.0 (100%). In the analysis template for the RED Method, a threshold of 0.9 for the Error metric will be used as the success condition. When HTTP faults are injected, the success rate is expected to decrease. If the success rate falls below 0.9, a rollback will be performed.

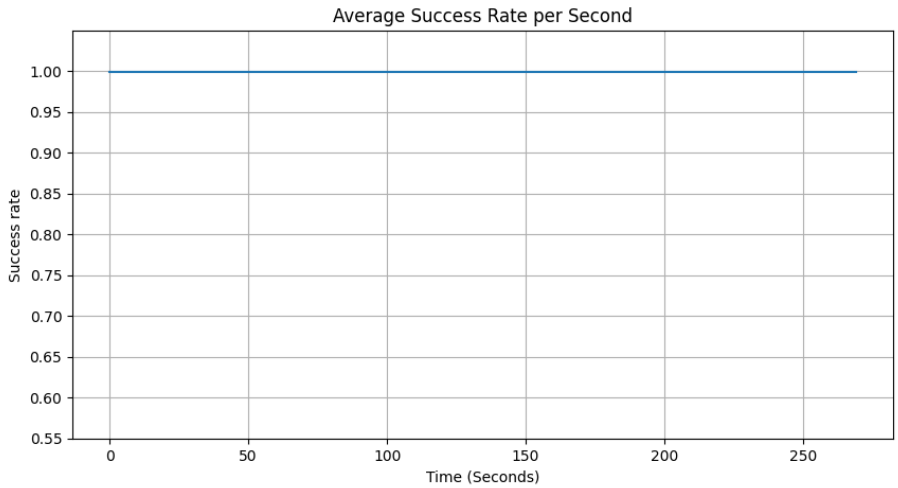

Figure 14: No fault injection, Success Rate (RED Method). The X-axis represents the time in seconds during the rollouts, and the Y-axis represents the success rate of requests not having HTTP 500.

Figure 15 shows the average duration per second from seven separate canary deployments of the Argo Proj application where no fault injection is present. The graph shows a low duration, little to none, due to no delay injection. The graph's duration fluctuates somewhat, but it is minimal. When delay injection is present, the duration is expected to increase. Thus, the analysis template for the RED Method will use a threshold of three seconds for the duration metric during the rollouts.

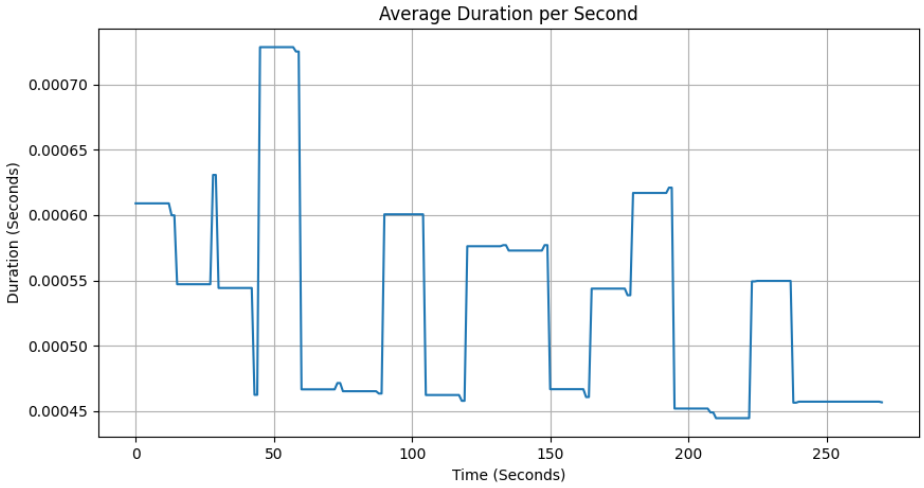

Figure 15: No fault injection, Duration (RED Method). The X-axis represents the time in seconds during the rollouts the Y-Axis represents the duration of time serving a request in seconds,

4 Result

In this chapter, a visualization of the results of the experiments mentioned in Section 3.3 is presented.

4.1 HTTP Fault Injection Experiment

This section presents the HTTP fault injection experiment results when performing seven canary deployments with Argo Rollouts using the frameworks *The Four Golden Signals* and *RED Method* in the analysis templates. During this experiment, HTTP 500 errors were injected into the canary version during deployment.

Figure 16 shows a graph of the average number of requests per second from the canary pod during the canary deployments. This graph represents the Traffic metric from the Four Golden Signals framework. From zero to approximately 60 seconds, there is a steady increase in the number of requests per second, starting at around eight and reaching a peak of about 23 requests per second. Between 60 and 100 seconds, the number of requests per second fluctuates slightly but generally hovers around 22 to 23 requests per second, indicating a stable period with minor variations. Around 100 seconds, the number of requests starts to decline heavily. At around 120 seconds, the analysis template signals a fault from the Success rate metric and waits 30 seconds before the rollback is performed. The rollback to the previous stable version is completed at around 150 seconds.

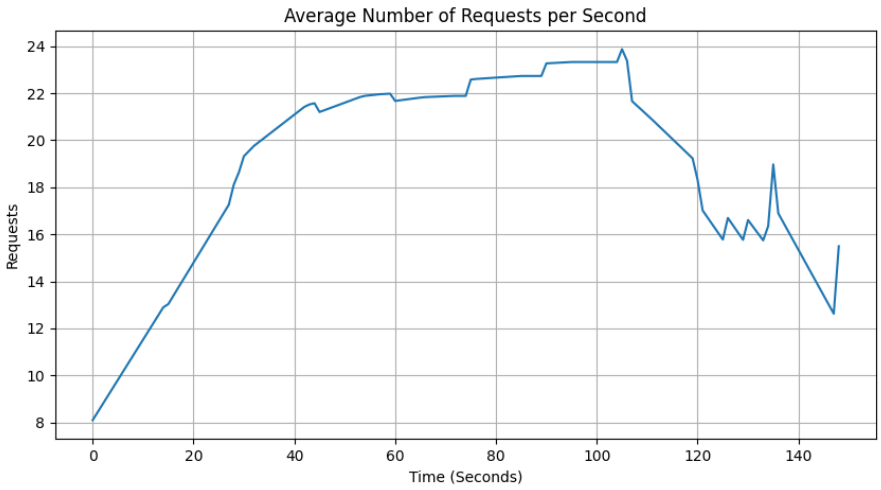

Figure 16: HTTP Fault Injection Experiment, Traffic (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents the number of requests per second, ranging from eight to 24.

Figure 17 shows a graph of the average success rate per second from the canary pod during the canary deployments. This graph represents the error metric from the Four Golden Signals framework. From zero to 60 seconds, the success rate starts at 1.00 and then steadily declines when more HTTP 500's are injected, dropping to 0.85 at 60 seconds. The success rate keeps on declining to 0.80 at around 120 seconds. This is when the analysis template signals a fault and then waits 30 seconds before a rollback is performed. The rollback to the previous stable version is completed at around 150 seconds.

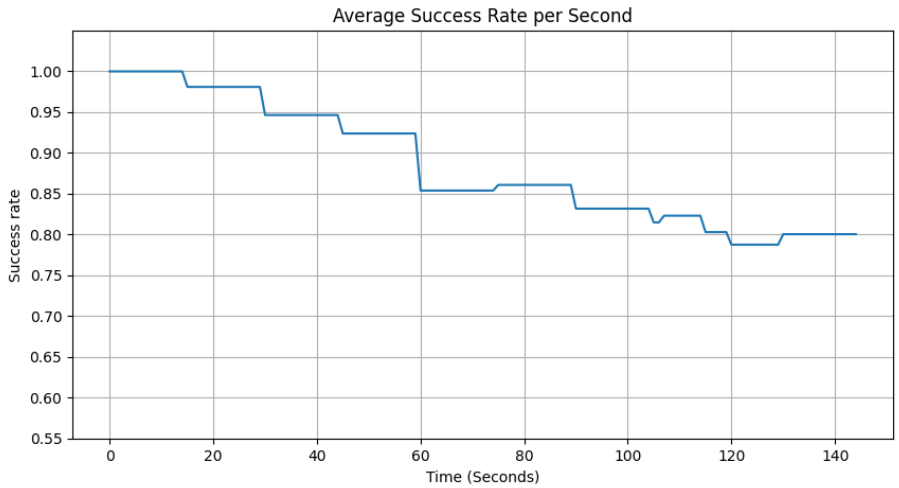

Figure 17: HTTP Fault Injection Experiment, Success rate (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents the success rate, ranging from 0.80 to 1.00.

Figure 18 shows the average latency per second from the canary pod during the canary deployments. This graph represents the latency metric from the Four Golden Signals framework. The latency is low during the whole experiment due to no delays being injected. At 120 seconds, the analysis template signals a fault in the success rate metric and waits 30 seconds before a rollback is performed. The rollback to the previous stable version is completed at around 150 seconds.

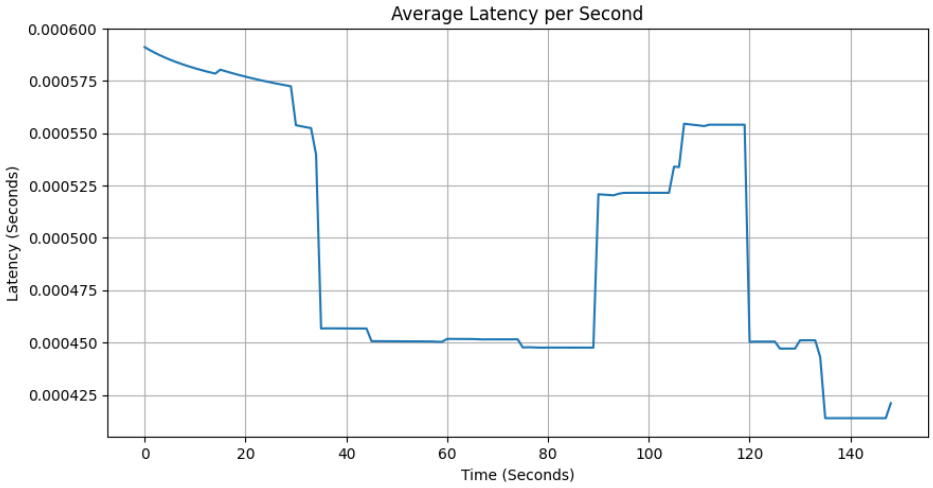

Figure 18: HTTP Fault Injection Experiment, Latency (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-Axis represents the latency in seconds, ranging from 0.000425 to 0.000600 seconds.

Figure 19 shows the average CPU usage per second from the canary pod during the canary deployments. This graph represents the saturation from the Four Golden Signals framework. From zero to approximately 40 seconds, there is a steady increase in CPU usage, starting at around 0.004 and reaching a peak of about 0.0085. Between 40 and 100 seconds, CPU usage fluctuates between 0.0075 and 0.009. At around 120 seconds, the analysis template signals a fault in the success rate metric and waits 30 seconds before a rollback is performed. The rollback to the previous stable version is completed at around 150 seconds.

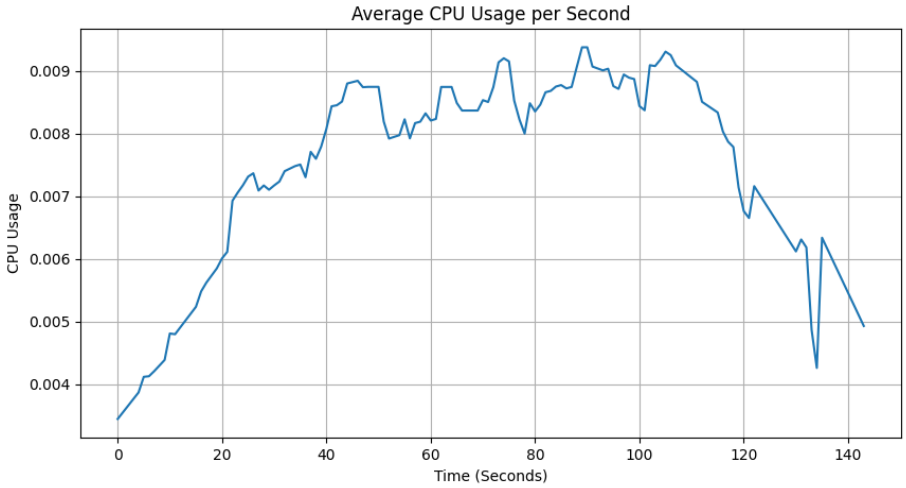

Figure 19: HTTP Fault Injection Experiment, Saturation (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents CPU usage, ranging from 0.004 to 0.009.

Figure 20 shows a graph of the average number of requests per second from the canary pod during the canary deployments. This graph represents the rate metric from the RED Method framework. From zero to approximately 40 seconds, there is a steady increase in the number of requests per second, starting at around eight and reaching a peak of about 16 requests per second. Between 40 and 100 seconds, the number of requests per second fluctuates, with values varying between 15 and 17 requests per second. At around 120 seconds, the analysis template signals a fault from the success rate metric and waits 30 seconds before the rollback is performed. The rollback to the previous stable version is completed at around 150 seconds.

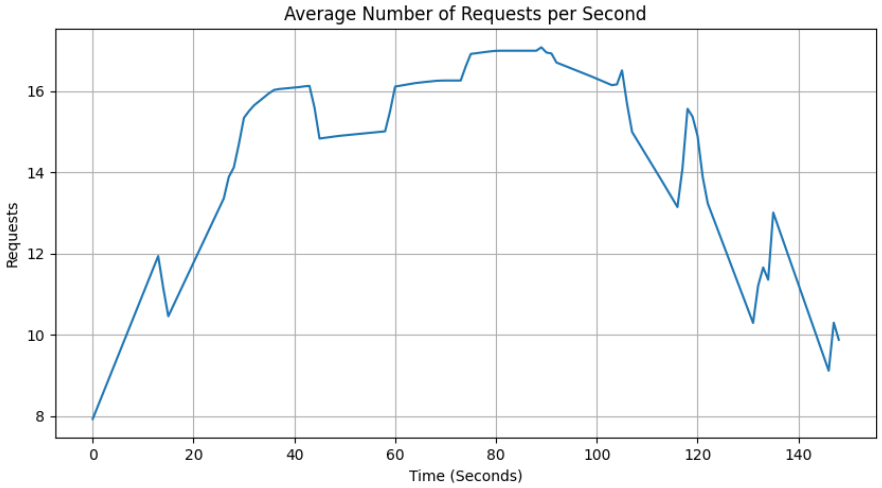

Figure 20: HTTP Fault Injection Experiment, Rate of requests (RED Method). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents the number of requests per second, ranging from 8 to 17.

Figure 21 shows a graph of the average success rate per second from the canary pod during the canary deployments. This graph represents the error metric from the RED Method framework.

It is important to note that the X-axis for the success rate graph ranges from zero to 130 seconds, unlike the other metrics, which range from zero to 150 seconds. This inconsistency was due to an unnoticed error in the data collection process. Despite this, the success rate data still provides valuable insights.

From zero to around 45 seconds, the success rate starts at 1.00 and then steadily declines, dropping to 0.90 at 45 seconds. The success rate keeps on declining until around 120 seconds and reaches a success rate of around 0.8. This is when the analysis template signals a fault and then waits 30 seconds before a rollback is performed. The rollback to the previous stable version is completed in around 150 seconds, but the graph does not show more than 130 seconds.

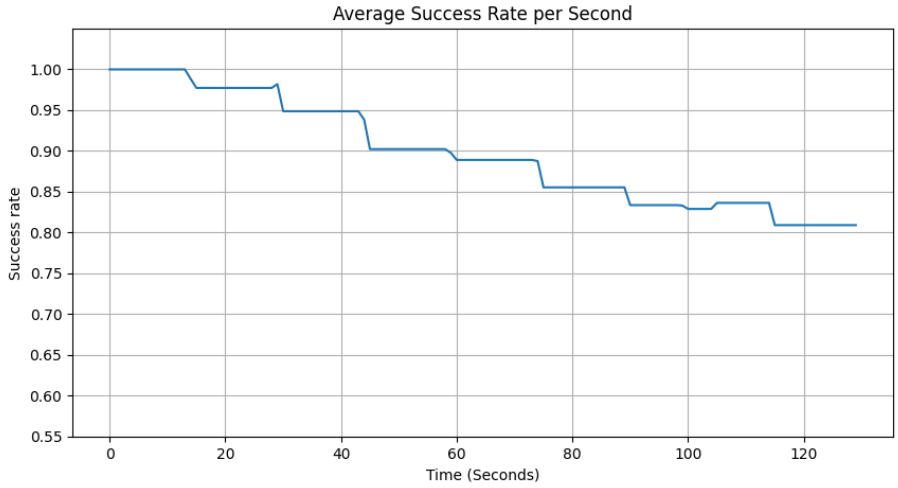

Figure 21: HTTP Fault Injection Experiment, Success rate (RED Method). The X-axis represents time in seconds, ranging from zero to 130 seconds. The Y-axis represents the success rate, ranging from 0.80 to 1.00.

Figure 22 shows a graph of the average duration per second from the canary pod during the canary deployments. This graph represents the duration metric from the RED Method framework. The duration is low throughout the experiment due to no delay injection. At 120 seconds, the analysis template signals a fault and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 150 seconds.

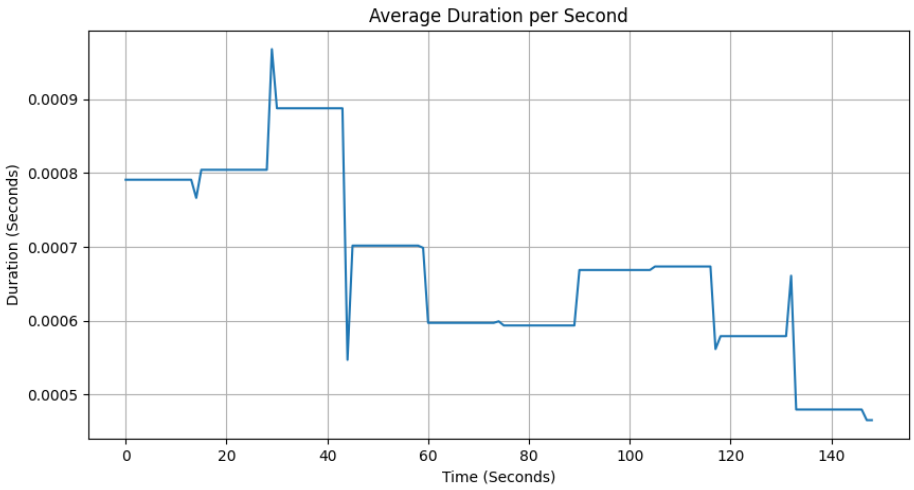

Figure 22: HTTP Fault Injection Experiment, Duration (RED Method). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents the average duration per request, ranging from 0.0005 to 0.0009 seconds.

4.2 Delay Injection

This section presents the results of the delay injection experiment when performing seven canary deployments with Argo Rollouts using the frameworks *The Four Golden Signals* and *RED Method* in the analysis templates. During this experiment, a delay was injected into the canary version during deployment.

Figure 23 shows a graph of the average number of requests per second from the canary pod during the canary deployments. This graph represents the Traffic metric from the Four Golden Signals framework. From zero to approximately 40 seconds, there's a steady increase in the number of requests per second, starting at around 10 and peaking at just above 20 requests per second. After that, the number of requests per second heavily decreases, and around 80 to 90 seconds, the number of requests per second is around one or two. At around 220 seconds, the analysis template signals a fault from the latency metric and waits 30 seconds before performing the rollback. The rollback to the previous stable version is completed at around 250 seconds.

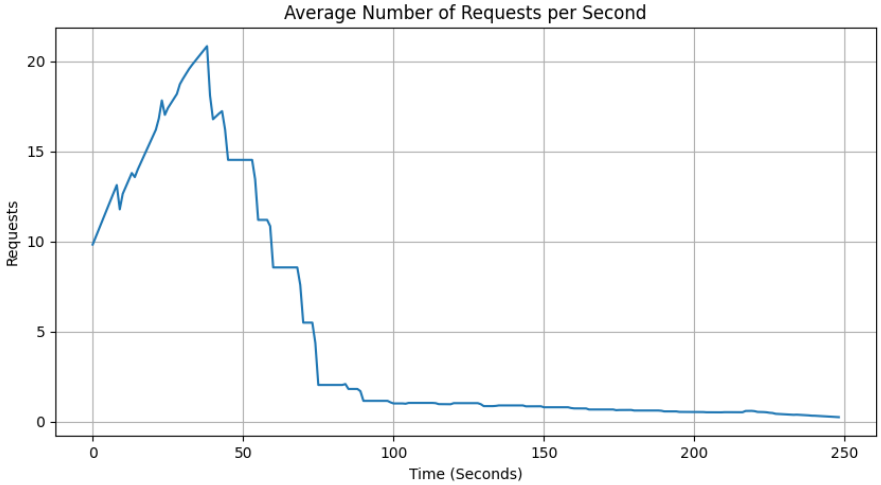

Figure 23: Delay Injection Experiment, Traffic (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to approximately 250 seconds. The Y-axis represents the number of requests per second, ranging from zero to 20

Figure 24 shows the average success rate from the canary pod during the canary deployments. This graph represents the error metric from the Four Golden Signals.

It is important to note that the X-axis for the success rate graph has a different range, unlike the other metrics, which range from zero to approximately 250 seconds. This inconsistency was due to an unnoticed error in the data collection process. Despite this, the success rate data still provides valuable insights.

During this experiment, no HTTP 500 errors were injected, and thus, the success rate remained constant at 1.00 throughout the entire period from zero to 250 seconds. At around 220 seconds, the analysis template signals a fault from the latency metric and waits 30

seconds before performing the rollback. The rollback to the previous stable version is completed at around 250 seconds.

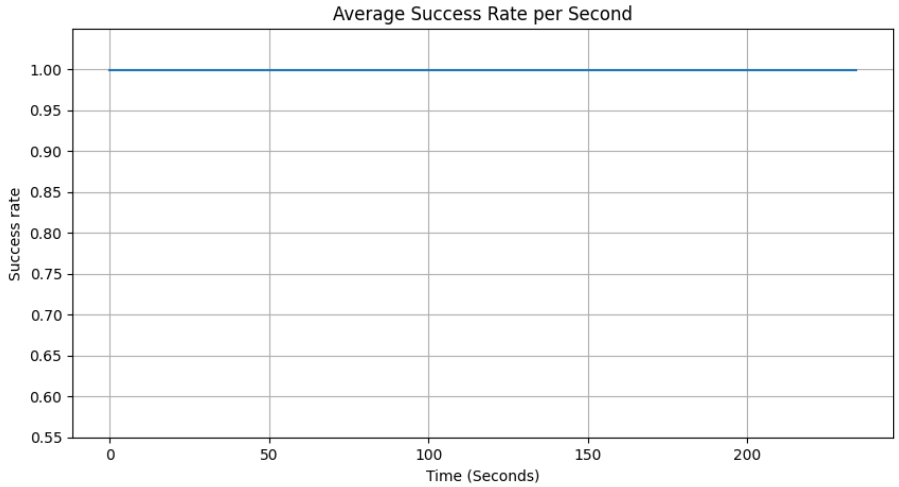

Figure 24: Delay Injection Experiment, Success Rate (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 250 seconds. The Y-axis represents the success rate.

Figure 25 shows the average latency per second from the canary pod during the canary deployments. This graph represents the latency metric from the Four Golden Signals framework. From zero to approximately 60 seconds, the latency increases by roughly one second. From 60 to 120 seconds, the latency increases to two seconds. At 180 seconds, the latency reaches three seconds and stops increasing. The analysis template signals for a fault in the latency metric at around 220 seconds, and the rollback is completed at around 250 seconds.

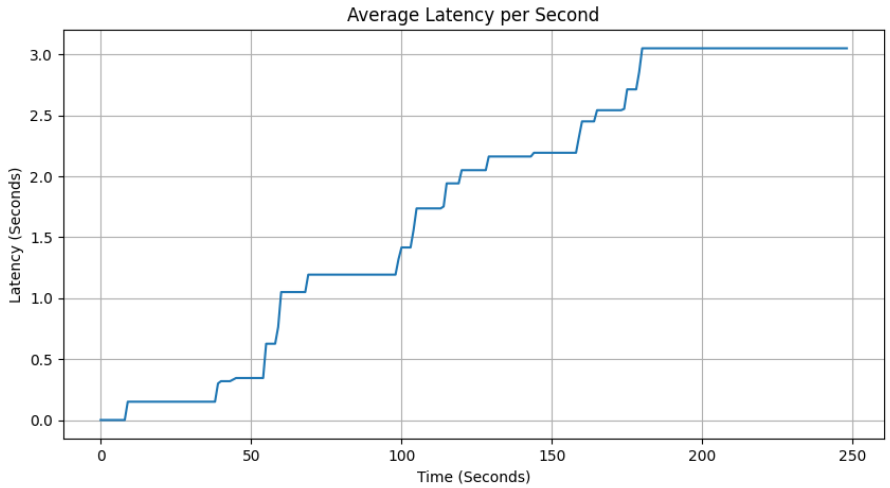

Figure 25: Delay Injection Latency (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 250 seconds. The Y-axis represents the average latency per request in seconds, ranging from zero to three seconds.

Figure 26 shows the average CPU usage per second from the canary pod during the canary deployments. This graph represents the saturation from the Four Golden Signals framework. From zero to approximately 50 seconds, there is a steady increase in CPU usage, peaking at slightly above 0.007 and then beginning to decline. This is around the same time as the latency increases to around 1 second. At 100 seconds, the CPU usage is down to about 0.001 and remains stable until the end. At around 220 seconds, the analysis template signals a fault from the latency metric and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 250 seconds.

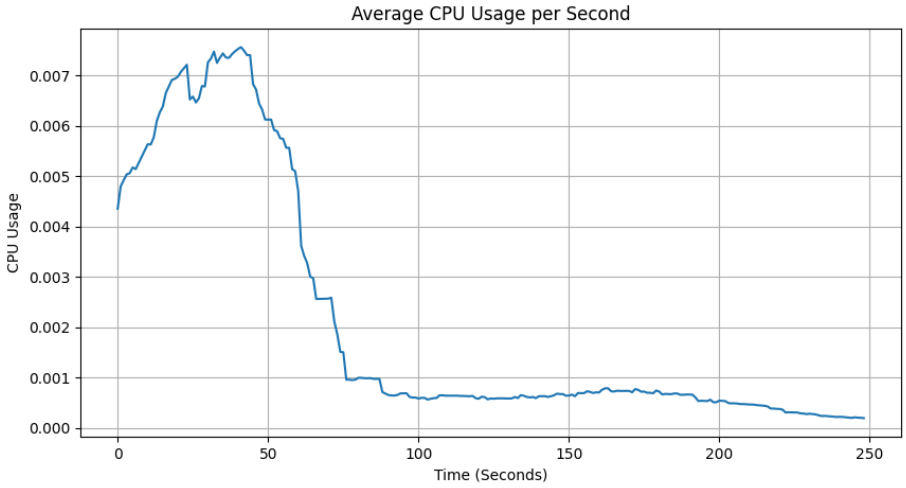

Figure 26: Delay Injection Experiment, Saturation (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 250 seconds. The Y-axis represents the average CPU usage per second.

Figure 27 shows a graph of the average number of requests per second from the canary pod during the canary deployments. This graph represents the rate metric from the RED Method framework. From zero to approximately 50 seconds, the number of requests per second steadily increases, starting at around 10, peaking at about 22 requests per second, and then beginning to decline. This is around the same time as the latency increases to around 1 second. The rate heavily declines to around two requests per second at 80 seconds and then steadily decreases. At 230 seconds, the analysis template signals for a fault in the duration metric and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 260 seconds.

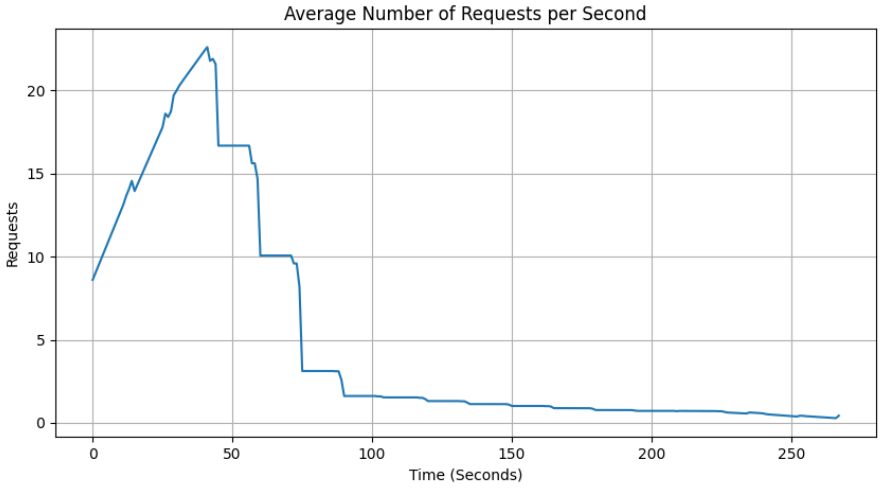

Figure 27: Delay Injection Experiment, Rate of requests (RED Method). The X-axis represents time in seconds, ranging from zero to 250 seconds. The Y-axis represents the number of requests per second, ranging from zero to about 22.

Figure 28 shows a graph of the average success rate per second from the canary pod during the canary deployments. This graph represents the error metric from the RED Method framework.

It is important to note that the X-axis for the success rate graph ranges from zero to around 250 seconds, unlike the other metrics, which range from zero to 260 seconds. This inconsistency was due to an unnoticed error in the data collection process. Despite this, the success rate data still provides valuable insights.

During this experiment, no HTTP 500 errors were injected, and thus, the success rate remained constant at 1.00 throughout the entire period from zero to 250 seconds. At around 230 seconds, the analysis template signals a fault from the duration metric, and Argo rollouts wait 30 seconds before performing the rollback. The rollback to the previous stable version is completed at around 260 seconds, but the graph does not show more than 250 seconds.

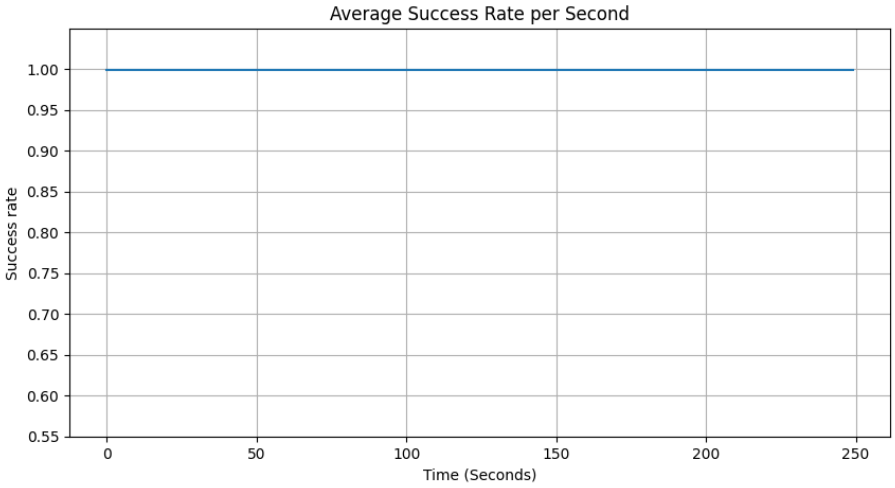

Figure 28: Delay Injection Experiment, Success rate (RED Method). The X-axis represents time in seconds, ranging from zero to 250 seconds. The Y-axis represents the success rate.

Figure 29 shows a graph of the average duration per second from the canary pod during the canary deployments. This graph represents the duration metric from the RED Method framework. From zero to approximately 40 seconds, the average duration per request remains low, gradually increasing from zero to around one second. At around 60 seconds, the duration increases from one second to around two seconds. The duration then remains stable at around two seconds until about 160 seconds. At 160 seconds, the duration increases again to around 3,5 seconds. Another step increase occurs at around 180 seconds, where the duration increases to around 4.5 seconds. The duration remains stable until the rollback, which is performed at around 260 seconds.

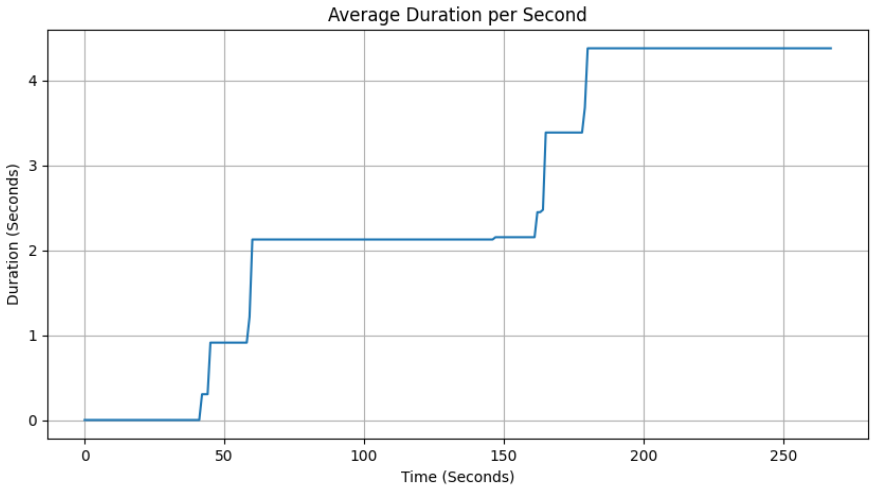

Figure 29: Delay Injection Experiment, Duration (RED Method). The X-axis represents time in seconds, ranging from zero to around 260 seconds. The y-axis represents the average duration per request in seconds, ranging from zero to 4.5 seconds.

4.3 Delay and HTTP Fault Injection

This section presents the delay and HTTP injection experiment results obtained using the frameworks *The Four Golden Signals* and *RED Method*. The Figures below illustrate the experiment's result, in which a delay and HTTP 500 errors were injected into the application during seven separate canary deployments.

Figure 30 shows a graph of the average number of requests per second from the canary pod during the canary deployments. This graph represents the Traffic metric from the Four Golden Signals framework. The graph starts at around 10 requests per second and slowly decreases due to the delay injection. At around 120 seconds, the analysis template signals a fault from either the Success rate or latency metric, and waits 30 seconds before performing the rollback. The rollback to the previous stable version is completed at around 150 seconds.

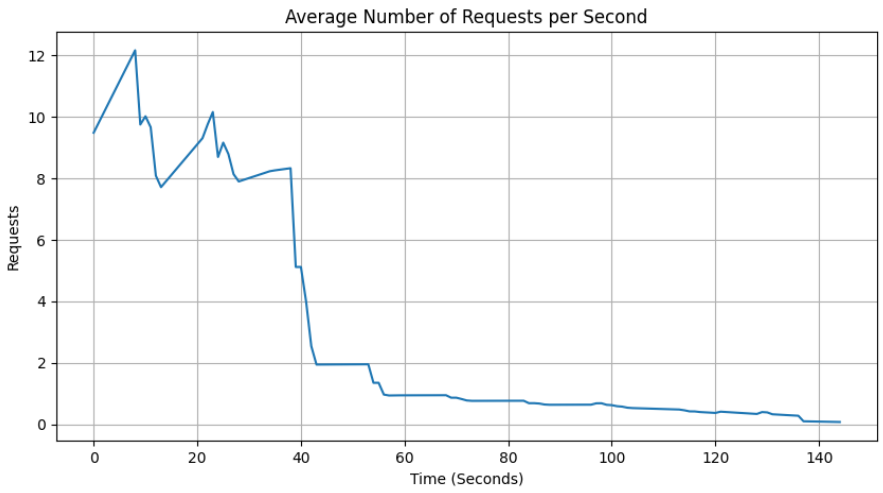

Figure 30: Delay and HTTP Fault Injection Experiment, Traffic (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents the number of requests, ranging from zero to 12.

Figure 31 shows the average success rate from the canary pod during the canary deployments. This graph represents the error metric from the Four Golden Signals.

It is important to note that the X-axis for the success rate graph has a different range, unlike the other metrics, which range from zero to approximately 150 seconds. This inconsistency was due to an unnoticed error in the data collection process. Despite this, the success rate data still provides valuable insights.

The success rate is steady from zero to around 25 seconds. Then, due to the HTTP 500, there is a small drop, and the success rate keeps on decreasing. From around 40 to 70 seconds, the success rate decreases to 0.95 and remains steady. At around 70 seconds, the success rate decreases to 0.85 and remains somewhat steady. At around 90 to 100 seconds, the success rate drops to 0.75. Around 120 seconds, the analysis template signals a fault. Argo Rollouts waits 30 seconds before a rollback is performed. The rollback to the previous stable version is completed in around 150 seconds, but the graph does not show more than 130 seconds.

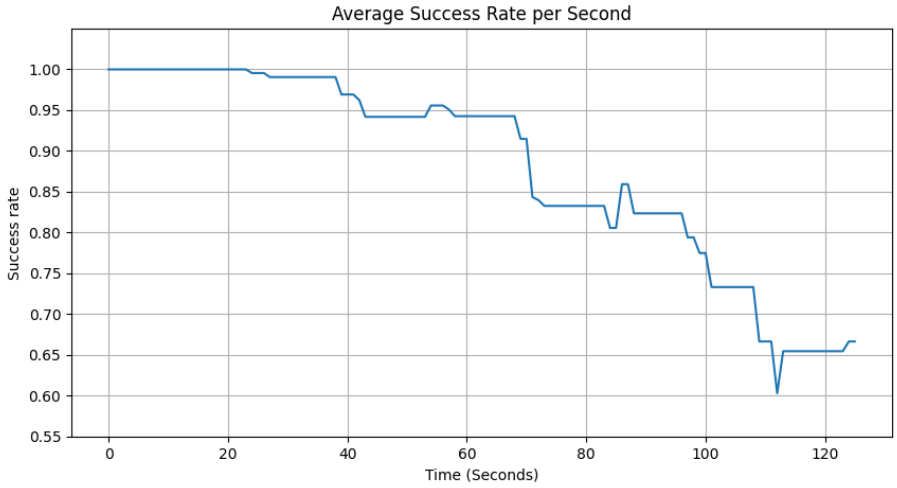

Figure 31: Delay and HTTP Fault Injection Experiment, Success Rate (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 130 seconds. The Y-axis represents the success rate.

Figure 32 shows the average latency per second from the canary pod during the canary deployments. This graph represents the latency metric from the Four Golden Signals framework.

The latency starts off at zero until around 10 seconds when the latency increases to around 0.75. At around 30 seconds, the latency reaches one second. Then, the increase in latency is repetitive. Every 10 seconds, it increases by 0.75, and every 30 seconds, the latency increases by 0.25 seconds. This pattern repeats for 90 seconds until the latency reaches three seconds. Then, it stops increasing. The analysis template signals for a fault at around 120 seconds, and the rollback is completed at around 150 seconds.

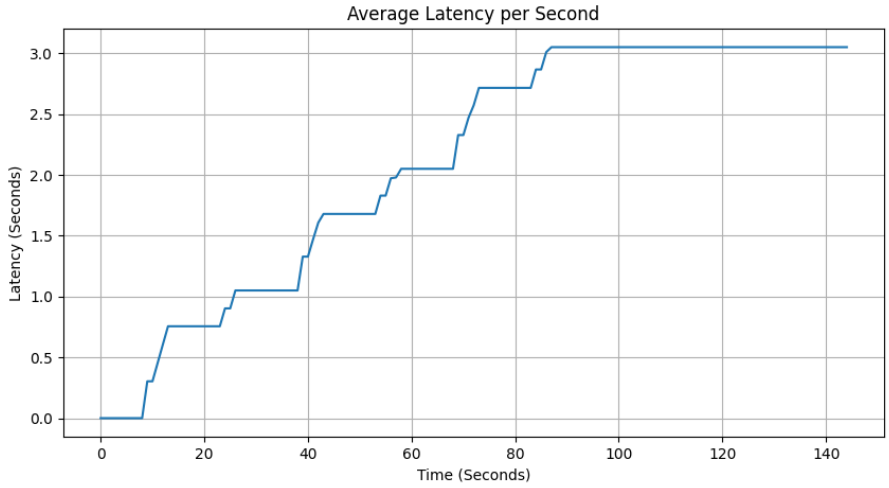

Figure 32: Delay and HTTP Fault Injection Experiment, Latency (The Four Golden Signals). The X-axis represents time in seconds, ranging from zero to 150 seconds. The Y-axis represents the average latency per request in seconds, ranging from zero to three seconds.

Figure 33 shows the average CPU usage per second from the canary pod during the canary deployments. This graph represents the saturation from the Four Golden Signals framework.

It is important to note that the X-axis for the saturation graph has a different range, unlike the other metrics, which range from zero to approximately 150 seconds. This inconsistency was due to an unnoticed error in the data collection process. Despite this, the saturation data still provides valuable insights.

From zero to approximately 20 seconds, there is a steady increase in CPU usage, peaking at around 0.0055 and then beginning to decline. This is around the same time as the latency increases to around one second. The CPU usage continues to decline, and at approximately 60 seconds, it reaches 0.0005, where it stops dropping heavily and decreases very slowly toward zero. At around 120 seconds, the analysis template signals a fault from the success rate or the latency metric and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 150 seconds, but the graph does not show more than approximately 138 seconds.

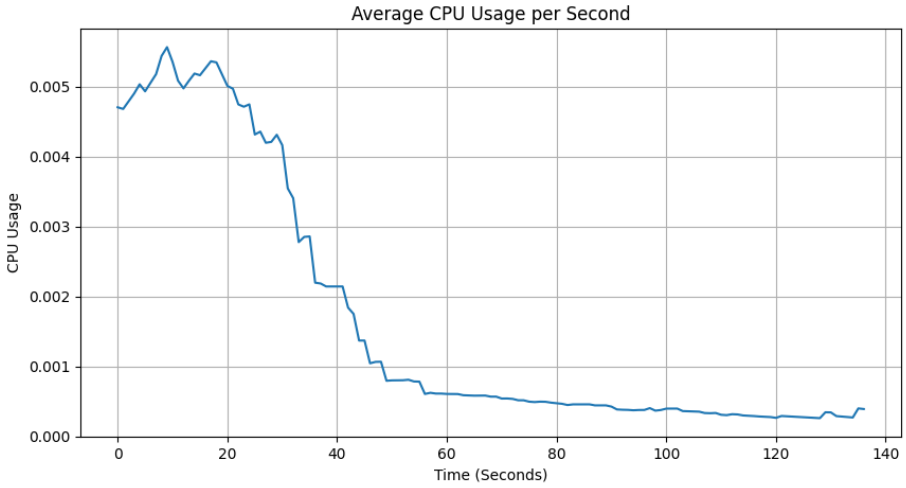

Figure 33: Delay and HTTP Fault Injection Experiment, Saturation (The four Golden Signals). The X-axis represents time in seconds, ranging from zero to around 140 seconds. The Y-axis represents the average CPU usage per second.

Figure 34 shows a graph of the average number of requests per second from the canary pod during the canary deployments. This graph represents the rate metric from the RED Method framework.

The rate starts at around seven requests per second, peaking at about 10 requests per second and then beginning to decline at around 40 seconds. This is around the same time as the delay increased to around 1 second. Due to the delay injection, the rate of requests heavily declines to around two requests per second and then steadily decreases. At around 135 seconds, the analysis template signals for a fault in either the success rate or duration metric and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 165 seconds.

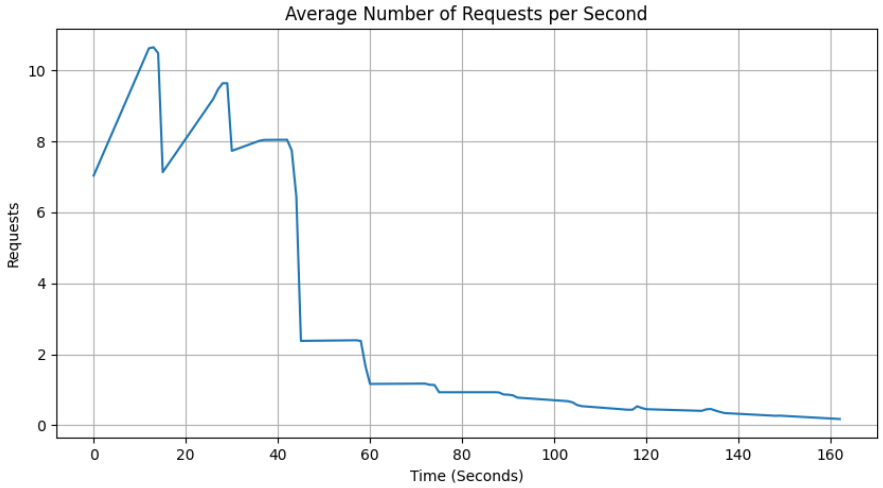

Figure 34: Delay and HTTP Fault Injection Experiment, Rate (RED Method). The X-axis represents time in seconds, ranging from zero to around 165 seconds. The Y-axis represents the number of requests, ranging from zero to 10 .

Figure 35 shows a graph of the average duration per second from the canary pod during the canary deployments. This graph represents the duration metric from the RED Method framework.

It is important to note that the X-axis for the success rate graph has a different range, unlike the other metrics, which range from zero to approximately 165 seconds. This inconsistency was due to an unnoticed error in the data collection process. Despite this, the saturation data still provides valuable insights.

Starting from zero to 20 seconds, the success rate is steady and does not show any decrement. The success rate drops Between 30 and 100 seconds due to the HTTP 500 injection, but it fluctuates and increases at certain times. This is due to a small offset in the data collection process with the timestamps. At around 135 seconds, the analysis template signals for a fault in either the success rate or duration metric and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 165 seconds, even though the graph stops at 145 seconds.

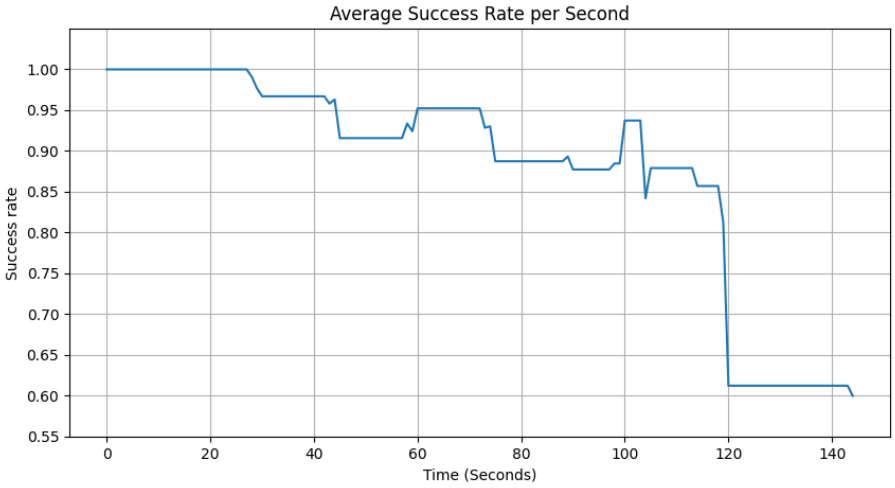

Figure 35: Delay and HTTP Fault Injection Experiment, Success rate (RED Method). The X-axis represents time in seconds, ranging from zero to around 145 seconds. The Y-axis represents the success rate.

Figure 36 shows a graph of the average duration per second from the canary pod during the canary deployments. This graph represents the duration metric from the RED Method framework. From zero to approximately 15 seconds, the duration goes up to about one second. At around 30 seconds, the delay has increased by one second, and the duration reaches about two seconds. Then it remains stable for around two seconds until 70 seconds, when it heavily increases to about 3.7. and at around 90 seconds, the duration reaches 4.5 seconds, which is where it will stay for the rest of the deployment. At around 135 seconds, the analysis template signals for a fault in either the success rate or duration metric and then waits 30 seconds before performing a rollback. The rollback to the previous stable version is completed at around 165 seconds.

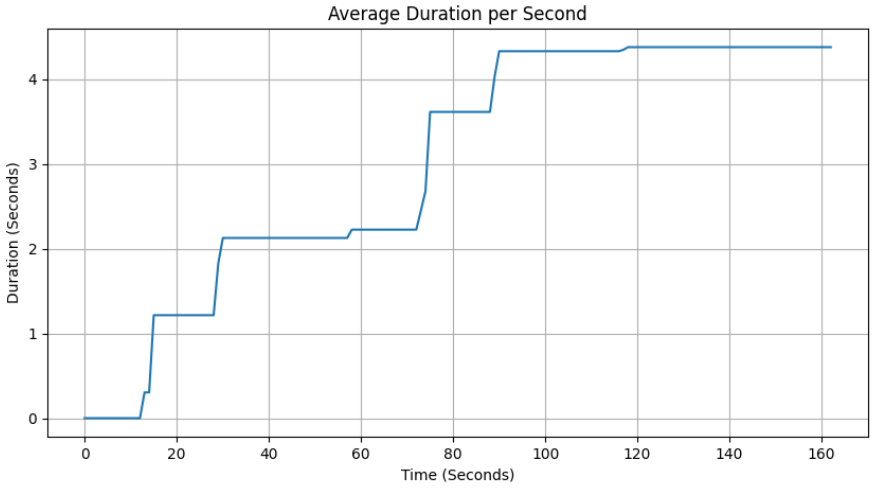

Figure 36: Delay and HTTP Fault Injection Experiment, Duration (RED Method). The X-axis represents time in seconds, ranging from zero to 165 seconds. The Y-axis represents the average duration per request in seconds, ranging from zero to 4.5 seconds.

5 Discussion

This chapter analyses and discuss the results from the experiments in Section 4 with respect to the research questions in Section 2.6.

5.1 Restate Research Questions

The primary objective of this study was to compare the performance of different monitoring frameworks (The Four Golden Signals and RED Method) when used with Argo Rollouts for automated deployments. The aim was to answer the following research questions:

- Which framework can be integrated with Argo Rollouts?
- How do these frameworks compare to each other, and how can they be used to automate the deployment procedure?
- How such automated deployments could impact the performance of applications?

5.2 Interpretation of Results

The results obtained from the HTTP Fault Injection, Delay Injection, and combined Delay and HTTP Fault Injection experiments provide key insights into the monitoring framework's comparative performance. This section discusses the results with respect to the research questions in Section 2.6.

RQ1: Which framework can be integrated together Argo Rollouts?

Both The Four Golden Signals and RED Method frameworks were successfully integrated with Argo Rollouts.

The Four Golden Signals framework was integrated through Prometheus metrics and custom analysis templates. The metrics monitored included traffic, success rate (errors), latency, and saturation (CPU Usage). Traffic measured the volume of requests processed over time, while success rate indicated the proportion of requests that did not return HTTP 500 errors. Latency measured the time taken to receive requests, and saturation measured the degree to which CPUs were utilized.

The RED Method framework was also integrated using Prometheus and custom analysis templates. The metrics monitored included rate, success rate (errors), and duration. Rate measured the number of requests processed over time, success rate indicated the proportion

of requests that did not return HTTP 500 errors, and duration captured the time taken to process requests.

The integration process for the frameworks involved configuring Prometheus queries to collect data for each metric. Custom analysis templates were created to evaluate the health of each deployment based on the metrics. Argo Rollouts used these templates to assess the health of canary deployments and decide whether to proceed with rollouts or initiate rollbacks. However, defining accurate thresholds for the metrics required careful calibration through multiple experimental runs. Additionally, optimizing Prometheus queries was essential for precise data collection.

In summary, both frameworks were successfully applied together with Argo Rollouts. The integration processes highlighted the importance of carefully calibrating metric thresholds to identify unhealthy deployments accurately.

RQ2: How do these frameworks compare to each other, and how can they be used to automate the deployment procedure?

The Four Golden Signals and the RED Method framework were compared in terms of the metrics they monitored, their effectiveness in identifying unhealthy deployments, and how they could be used to automate the deployment procedure.

The Four Golden Signals framework provided detailed insights into the health and performance of the application by monitoring four key metrics: traffic, success rate (errors), latency, and saturation (CPU usage). The traffic metric measured the volume of incoming requests, while the success rate metric monitored the proportion of requests that were successfully processed without HTTP 500 errors. Latency provided a measure of the time taken to process requests, and saturation gauged the degree to which system resources (CPU) were utilized. These metrics allowed for comprehensive monitoring of the application's behavior, particularly in identifying unhealthy canary deployments. The latency and success rate metrics were particularly effective in signaling potential issues during the experiments.

The RED Method framework, on the other hand, focused on three key metrics: rate, success rate (errors), and duration. The rate metric measured the number of requests processed per second, providing a sense of the system throughput. Errors monitored the proportion of failed requests, and duration measured the time taken to process requests. Although the RED Method involved fewer metrics than the Four Golden Signals framework, it effectively identified unhealthy deployments during the experiments.

When comparing the two frameworks, the Four Golden Signals provided more comprehensive insights due to the inclusion of saturation and latency. Saturation offered a unique perspective on system resource utilization that the RED Method did not capture, while latency provides a finer granularity for detecting slow-performing requests. However, the RED Method was simpler to configure and interpret because of the fewer metrics.

In terms of automating the deployment procedure, both frameworks could be effectively integrated with Argo Rollouts. Custom analysis templates were created for each framework, and Prometheus was used to collect and query the relevant metrics. The analysis templates defined the thresholds for each metric that would trigger a rollback if exceeded. If any metric indicated an unhealthy deployment, a rollback was triggered.

In conclusion, while the Four Golden Signals provides a more comprehensive monitoring solution due to the inclusion of saturation and latency metrics, the RED Method's simplicity made it easier to configure and interpret. Both frameworks effectively automated the deployment procedure with Argo Rollouts, ensuring reliable and controlled rollouts. Additional metrics related to saturation, such as memory and network bandwidth, could be included to improve effectiveness with the Four Golden Signals.

RQ3: How such automated deployments could impact the performance of applications?

The impact of automated deployments on application performance varied depending on the type of fault injection used during the experiments. The effects were analyzed through three different experiments: HTTP Fault Injection, Delay Injection, and Combined Fault Injection (HTTP Fault injection and Delay injection).

In the HTTP Fault Injection experiment, HTTP 500 errors were injected into the application during a canary deployment. Both frameworks accurately detected unhealthy deployments by monitoring success rate (errors). The Four Golden Signals framework used the success rate metric to identify the drop in successfully processed requests, triggering a rollback when the success rate fell below a predefined threshold. Similarly, the RED Method framework relied on the same metric to detect errors and initiate a rollback. The other metrics, such as Traffic, latency, duration, and saturation, did not show any signs of trouble and stayed relatively accurate to the normal behavior.

The delay injection experiment introduced delays to the application during a canary deployment. Both frameworks detected performance issues through their respective latency and duration metrics. In the Four Golden Signals framework, the latency metric showed a clear increase in the time taken to process requests, prompting a rollback when the latency exceeded the threshold. The RED Method framework used the duration metric to reveal the same performance issue.

There is a slight difference in the results of the RED Method framework and the Four Golden Signals when comparing duration and latency shown in Section 4.2. When injecting the application with a delay of one second, which increases every minute. It is important not to confuse these two metrics with each other because otherwise, the rollback would be performed in the wrong situation.

Traffic and rate metrics had a reduction in requests due to the delays. Saturation revealed a similar pattern in CPU utilization as the application struggled to handle the delays.

In the Combined Fault Injection experiment, both delays and HTTP 500 errors were injected simultaneously. Both frameworks accurately detected unhealthy deployments through a combination of success rate (errors) and latency/duration metrics. In the Four Golden Signals framework, the success rate metric revealed a significant drop in successfully processed requests, while the latency metric showed a notable increase. These signals prompted a rollback.

In the RED Method framework, success rate (errors) and duration metrics revealed similar trends, leading to a rollback. The rate metric showed reduced throughput due to the impact of delays.

In summary, automated deployments using Argo Rollouts impacted application performance differently depending on the type of fault injection. Both frameworks accurately identified unhealthy deployments and ensured rollbacks, preventing further performance degradation.

6 Conclusion

This chapter discusses whether the aims and objectives have been achieved, reflects on the technical challenges encountered, and provides personal reflections on the work during the thesis. It also suggests areas for future work.

6.1 Aims and Objectives

In my thesis, I set aims and objectives to guide the direction of my research and ensure a structured approach. These aims and objectives served as milestones, and I have addressed each one throughout my work.

I aimed to create an experimental environment by setting up a lightweight Kubernetes using MicroK8s, which aligns with the requirement for a lightweight setup. Additionally, I used a sample application provided by the Argo project for experimentation within the Kubernetes environment. The setup of Argo Rollout was completed successfully, ensuring that all the necessary tools were in place.

Secondly, I investigated different frameworks. I explored how the Four Golden Signals and RED Method frameworks could be integrated with Argo Rollout. Furthermore, I delved into how to implement rollouts and rollbacks based on predefined thresholds from the metrics provided by these frameworks.

Lastly, I aimed to evaluate the impact of automated deployments on application performance, availability, and reliability. To do this, I selected appropriate evaluation criteria and conducted a series of experiments. These experiments included HTTP Fault Injection, Delay Injection, and a combination of both. By analyzing the results, I was able to evaluate the performance of the applications under various fault conditions. The data collected from these experiments confirmed the objectives of analyzing and evaluating the impact of automated deployments.

6.2 Technical Issues

During the project, a few technical issues were encountered. The first technical issue that occurred was during the initial phase of setting up and exposing the different services in Kubernetes through the Google Cloud platform. Because of proxys and firewall rules the cluster could not be reached outside the VM. With help of the employees at Elastisys this issue got fixed relatively quickly even though it stalled the project for a few days, due to not knowing if the problem was because of GCP or the cluster I had set up.

After successfully installing and setting up the cluster, another problem emerged with the

node experiencing disk pressure. This issue made the node extremely slow and eventually inaccessible. The root cause of this problem was the insufficient storage and RAM available on the node. To avoid this problem in the future, an upgrade was made to the node's specifications.

Another technical issue encountered during the project was the first choice of application, Google Botique. During the phase of implementing fault injection into the application, I faced several issues. One issue with the application was that it was hard to implement fault injection into the application itself. I also tried using errors via Istio fault injection in the destination rule/ virtual services. There were many errors, and the fault injection was not working as intended. This caused a lot of delay with the project, and I had to take the option of using another application in order to not fall behind in schedule. This was a huge let down because the application, Google Botique is a very good application that can simulate a real application running on Kubernetes. I solved the issue by using the ArgoProj application that they provide for demonstration purposes.

A minor issue that was found very late was the time interval in the X-axis of some success rate plots used for the results. Due to an error in the Prometheus query, the time interval was set to the wrong value. Causing the time interval to be shorter. This is actually a minor problem because just a few seconds were removed from the plot. If I were to redo the experiments, the plot would have looked the same and had the same time interval in the X-axis as the other plots.

6.3 Personal Reflection

I am happy and proud of my work on this thesis project. If I could restart the project today, I would have distributed my time differently. Although I followed the time schedule well enough, I could have started the practical part of the project earlier. At the start of the project, I was very focused on writing as much as possible and underestimated the practical work of setting up and adjusting the different tools.

In hindsight, I would have researched more about the practical aspects of the project and less about the theoretical part because my way of learning is by doing. Additionally, the subject of the thesis was very broad at the start, and I initially thought that I just had to find challenges and opportunities related to Argo Rollouts, which was the original context of the thesis. However, this made me realize that I needed to narrow down the thesis towards monitoring frameworks and their integration with Argo Rollouts.

Furthermore, I found it challenging to find related work on Argo Rollouts due to the tool being relatively new. Instead I found work that I could draw parallels towards.

Despite this, I am proud of my accomplishment and I have learned a lot during this semester from myself, my supervisor, and Elastisys. The most significant improvement I have made is feeling more comfortable using Kubernetes.

6.4 Future Work

The future work section will bring up some interesting suggestions on fields that would be interesting to investigate further. During this thesis, Blue-green Deployment was not experimented on. The future work would involve investigating the Blue-green Deployment strategy using Argo Rollouts and comparing its effectiveness, reliability, and performance with other deployment strategies like canary deployments.

Due to cyberattacks becoming more common, some businesses need to reinforce their security. Further research could be conducted to investigate the robustness of the system and use more advanced fault injection scenarios, including security breaches or other types of attacks during a deployment with Argo Rollout.

Businesses can benefit from researching the cost savings achieved by using Argo Rollouts over a period of time. The research could uncover significant reductions in infrastructure, operational, and downtime costs. This information would provide a compelling financial case for implementing automated deployment strategies.

Another future work could involve adapting Argo Rollouts for integration with more tools, such as CI/CD pipelines and various DevOps frameworks. This could enhance automation, streamline deployments, and improve the overall efficiency of software delivery processes.

References

[1] Rancher Admin. RED Method for Prometheus – 3 Key Metrics for Monitoring. https://www.suse.com/c/rancher$_b log/red-method-for-prometheus-3-key-metrics-for-monitoring/.(visited 2024-02-28).

[2] Agilemanifesto. Manifesto for Agile Software Development. https://agilemanifesto.org/. (visited 2024-02-14).

[3] Apache. What is Mesos? A distributed systems kernel. https://mesos.apache.org/documentation/latest/. (visited 2024-04-16).

[4] Argoproj. Concepts. https://argoproj.github.io/argo-rollouts/concepts/rolling-update. (visited 2024-02-08).

[5] Argoproj. rollouts-demo. https://github.com/argoproj/argo-rollouts. (visited 2024-04-17).

[6] Argoprojects. Argo Rollouts - Kubernetes Progressive Delivery Controller. https://argoproj.github.io/argo-rollouts/. (visited 2024-02-23).

[7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[8] David Chou. Cloud Service Models (IaaS, PaaS, SaaS) Diagram. https://dachou.github.io/2018/09/28/cloud-service-models.html. (visited 2024-01-31).

[9] Google Cloud. State-of-DevOps-2021. https://services.google.com/fh/files/misc/state-of-devops-2021.pdf. (visited 2024-05-10).

[10] codefresh. CI/CD and Agile: Why CI/CD Promotes True Agile Development. https://codefresh.io/learn/ci-cd-pipelines/ci-cd-and-agile-why-ci-cd-promotes-true-agile-development/. (visited 2024-02-17).

[11] Eric Conrad. Waterfall Model. https://www.sciencedirect.com/topics/computer-science/waterfall-model. (visited 2024-04-16).

[12] Docker. Swarm mode overview. https://docs.docker.com/engine/swarm/. (visited 2024-04-16).

[13] Mayank Gokarna and Raju Singh. Devops: a historical review and future works. In *2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 366–371. IEEE, 2021.

[14] Google. What is Cloud Computing? https://cloud.google.com/learn/what-is-cloud-computing. (visited 2024-01-31).

[15] Brendan Gregg. The USE Method. https://www.brendangregg.com/usemethod.html. (visited 2024-02-28).

[16] IBM. What is Cloud Computing? https://www.ibm.com/topics/cloud-computing. (visited 2024-01-31).

[17] IBM. What is containerization. https://www.ibm.com/topics/containerization. (visited 2024-02-07).

[18] IBM. What is Virtualization. https://www.ibm.com/topics/virtualization. (visited 2024-02-06).

[19] Bilgin Ibryam and Roland Huß. *Kubernetes Patterns*. " O'Reilly Media, Inc.", 2022.

[20] INAP. What are the Differences Between IaaS, PaaS and SaaS. https://www.inap.com/blog/iaas-paas-saas-differences/. (visited 2024-02-06).

[21] Istio. Getting started. https://istio.io/latest/docs/setup/getting-started/download. (visited 2024-05-03).

[22] Istio. Istio. https://istio.io/. (visited 2024-05-12).

[23] William Johansson. A Comparison of CI/CD tools on Kubernetes, 2022.

[24] Tech Target Kate Brush. Agile software development. https://www.techtarget.com/searchsoftwarequality/definition/agile-software-development. (visited 2024-02-14).

[25] Gene Kim, Kevin Behr, and Kim Spafford. *The phoenix project: A novel about IT, DevOps, and helping your business win*. IT Revolution, 2014.

[26] Kubernetes. Kubernetes. https://kubernetes.io/. (visited 2024-05-12).

[27] T Kubernetes. Kubernetes. *Kubernetes. Retrieved May*, 24:2019, 2019.

[28] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[29] MicroK8s. Get started. https://microk8s.io/docs/getting-started. (visited 2024-05-01).

[30] MicroK8s. MicroK8s. https://microk8s.io/. (visited 2024-05-01).

[31] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[32] Sten Pittet. Continuous integration vs. delivery vs. deployment. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. (visited 2024-02-22).

[33] Prometheus. Prometheus. https://prometheus.io/. (visited 2024-05-12).

[34] Redhat. What is CI/CD. https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[35] RedHat. What is container orchestration? https://www.redhat.com/en/topics/containers/what-is-container-orchestration. (visited 2024-02-09).

[36] RedHat. What's a service mesh? https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh. (visited 2024-05-12).

[37] Chaitanya K Rudrabhatla. Comparison of zero downtime based deployment techniques in public cloud infrastructure. In *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 1082–1086. IEEE, 2020.

[38] Nichil Strasser. *An Evaluation of Canary Deployment Tools*. PhD thesis, University of Applied Sciences, 2023.

[39] Gustaf Söderlund. Git Repository. https://github.com/Sudda-king/masterthesis. (visited 2024-05-01).

[40] Devops Toolkit. Progressive Delivery Explained - Big Bang (Recreate), Blue-Green, Rolling Updates, Canaries. https://www.youtube.com/watch?v=HKkhD6nokC8. (visited 2024-02-08).

[41] Billy Yuen, Alexander Matyushentsev, Todd Ekenstam, and Jesse Suen. *GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux*. Simon and Schuster, 2021.