Degree Project in Computer Science and Engineering

Second cycle, 30 credits

The Design and Implementation of Automated Deployment Pipelines for Amazon Web Services

GitOps practices in the context of CI/CD pipelines using GitLab and Infrastructure as Code

MANISH KUMAR

The Design and Implementation of Automated Deployment Pipelines for Amazon Web Services

GitOps practices in the context of CI/CD pipelines using GitLab and Infrastructure as Code

MANISH KUMAR

Master's Programme, ICT Innovation, 120 credits
Date: June 13, 2024

Supervisors: Rajendra Shivaji Patil, Subrata Bauri
Examiner: Viktoria Fodor
School of Electrical Engineering and Computer Science
Host company: Scania CV AB
Swedish title: Utformningen och implementeringen av automatiserade distributionspipelines för Amazon Web Services
Swedish subtitle: GitOps-praktiserar i samband med CI/CD-pipelines med GitLab och Infrastructure som kod

Abstract

The integration of GitOps and Infrastructure as Code within the context of continuous integration and continuous delivery/deployment (CI/CD) pipelines represents a significant challenge in software and infrastructure deployment, particularly in public cloud environments. This thesis provides a comprehensive analysis and investigates how such integrations in Amazon Web Services can improve deployment efficiency, reliability in software and infrastructure delivery, and security while addressing the gaps in the existing Development and Operations (DevOps) practices. The problem is that, in public cloud platforms, the complexity of deploying infrastructure using automated pipelines often results in a lack of version control, inefficiency or security loopholes. GitOps concepts and Infrastructure as Code principles have facilitated ease of deployment for applications and infrastructure, but there is still room for improvement in understanding their effects when combined with automated pipelines. This gap delays the optimization of best practices in deployment strategies and the full potential of infrastructure deployment processes in the public cloud, making it an essential area for academic research.

This thesis utilizes a systematic approach, implementing a controlled GitOps environment using GitLab and Amazon Web Services that integrate GitOps concepts and Infrastructure as Code principles with Development, Security and Operations (DevSecOps) practices. This thesis analyzes the effects of GitOps and Infrastructure as Code on deployment efficiency and reliability through automated pipelines built on GitLab and provides best practices for improving efficiency, reliability, and security. These findings offer significant insights for organizations to refine their software development and infrastructure deployment practices. Furthermore, this thesis provides practitioners with empirical evidence to guide the adoption of GitOps and Infrastructure as Code with automated pipelines in software development and infrastructure deployment practices.

Keywords

GitOps, Infrastructure as Code, GitLab, DevOps, Docker Containers, Kubernetes, Git, Continuous Integration and Continuous Delivery/Deployment, Amazon Web Services, Terraform, Pulumi, Argo CD

Sammanfattning

Integrationen av GitOps och Infrastructure som kod inom ramen för kontinuerlig integration och kontinuerlig leverans/deployment (CI/CD) pipelines representerar en betydande utmaning i programvaru- och infrastrukturdistribution, särskilt i offentliga molnmiljöer. Den här avhandlingen ger en omfattande analys och undersöker hur sådana integrationer i Amazon Web Services kan förbättra implementeringseffektiviteten, tillförlitligheten i leverans av programvara och infrastruktur och säkerhet samtidigt som man åtgärdar luckorna i de befintliga utvecklings- och driftrutinerna (DevOps). Problemet är att i offentliga molnplattformar resulterar komplexiteten i att distribuera infrastruktur med hjälp av automatiserade pipelines ofta i brist på versionskontroll, ineffektivitet eller kryphål i säkerheten. GitOps-koncept och Infrastructure as Code-principer har underlättat installationen för applikationer och infrastruktur, men det finns fortfarande utrymme för förbättringar när det gäller att förstå deras effekter när de kombineras med automatiserade pipelines. Denna lucka fördröjer optimeringen av bästa praxis i distributionsstrategier och den fulla potentialen av infrastrukturimplementeringsprocesser i det offentliga molnet, vilket gör det till ett viktigt område för akademisk forskning.

Detta examensarbete använder ett systematiskt tillvägagångssätt och implementerar en kontrollerad GitOps-miljö med hjälp av GitLab och Amazon Web Services som integrerar GitOps-koncept och infrastruktur som kodprinciper med utveckling, säkerhet och drift (DevSecOps) praxis. Den här avhandlingen analyserar effekterna av GitOps och Infrastructure som kod på implementeringseffektivitet och tillförlitlighet genom automatiserade pipelines byggda på GitLab och ger bästa praxis för att förbättra effektivitet, tillförlitlighet och säkerhet. Dessa resultat ger betydande insikter för organisationer att förfina sina metoder för programvaruutveckling och infrastrukturinstallation. Dessutom förser denna avhandling praktiker med empiriska bevis för att vägleda antagandet av GitOps och Infrastructure som kod med automatiserade pipelines i mjukvaruutveckling och infrastrukturimplementeringsmetoder.

Nyckelord

GitOps, Infrastruktur som kod, GitLab, DevOps, Docker-containrar, Kubernetes, Git, Kontinuerlig Integration och Kontinuerlig Leverans/Deployment, Amazon Web Services, Terraform, Pulumi, Argo CD

Acknowledgments

I would like to extend my deepest gratitude to Subrata Bauri, my industrial supervisor at Scania CV AB, where I conducted my research. He provided me with assistance in a number of ways during the research. He helped me solve several configuration problems when the research was technically implemented. In addition, I want to thank Johannes Slettengren, my reporting manager at Scania CV AB. He warmly welcomed me when I joined his Scania team and supported me while I conducted my research. I also extend my gratitude to the Scania team members in the Autonomous Development department for their encouragement and shared curiosity.

I also want to express my gratitude to Dr Rajendra Shivaji Patil at KTH Royal Institute of Technology, who oversaw my thesis and helped me plan and organize the research. He was enormously helpful in providing insightful answers to all of my questions. Furthermore, I express my profound gratitude to Professor Viktoria Fodor, my examiner at KTH Royal Institute of Technology, for her invaluable input and assistance in helping me comprehend the requirements of the degree project and for providing valuable feedback throughout the research phase.

Stockholm, June 2024
Manish Kumar

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	Application programming interface
AWS	Amazon Web Services
CI/CD	Continuous Integration and Continuous Delivery/Deployment
DevOps	Development and Operations
DevSecOps	Development, Security and Operations
DinD	Docker in Docker
DSL	Domain Specific Language
EC2	Elastic Compute Cloud
EKS	Amazon Elastic Kubernetes Service
GIT	Global Information Tracker
HCL	HashiCorp Configuration Language
IaC	Infrastructure as Code
K8s	Kubernetes
NIST	National Institute of Standards and Technology
OWASP	Open Worldwide Application Security Project
PaaS	Platform as a service
R/W	Read and Write
SSH	Secure Shell
VCS	Version control system
VPC	Virtual private cloud
YAML	Yet another markup language

List of Symbols Used

The following symbols will be later used within the body of the thesis.

$\bar{X}_1$ Mean of the first sample, see equation (B.1), ... page 98

$\bar{X}_2$ Mean of the second sample, see equation (B.1), ... page 98

μ_1 Mean of the first population, see equation (B.1), ... page 98

μ_2 Mean of the second population, see equation (B.1), ... page 98

n_1 Sample size of the first group, see equation (B.2), ... page 99

n_2 Sample size of the second group, see equation (B.2), ... page 99

S_1 Sample standard deviation of the first group, see equation (B.2), page 99

S_2 Sample standard deviation of the second group, see equation (B.2), page 99

S_p Pooled estimate of common standard deviation, see equation (B.2), page 99

Z Critical value for constructing confidence intervals, see equation (B.3), page 99

Chapter 1

Introduction

Most businesses are moving their business workloads to public cloud platforms due to their benefits, such as scalability and flexibility. Utilizing public cloud services assists companies in reducing their dependency on physical servers and infrastructure in data centers, thereby cutting down on maintenance and upfront costs. As a result, the widespread adoption of public cloud platforms has significantly influenced how businesses manage software development and infrastructure deployment. This adoption has led to the exploration of new approaches to improving these processes. One such approach is GitOps, which builds on Development and Operations (DevOps) principles to automate infrastructure deployment. It is an operational framework that applies DevOps practices such as Version control system (VCS), code collaboration, and Continuous Integration and Continuous Delivery/Deployment (CI/CD) and applies them to automate infrastructure deployment. This thesis examines three important concepts utilized in the software development industries: CI/CD, Infrastructure as Code (IaC) and GitOps. These methods are revolutionizing industries where software development and infrastructure deployment are essential to their operations. Moreover, GitOps demonstrates an improvement in managing the applications and the infrastructure with the help of Global Information Tracker (GIT) repositories, which serve as the single source of truth for infrastructure provisioning. Likewise, IaC tools automate infrastructure provisioning through code, which facilitates reproducibility, scalability, and consistency.

In real-work software development scenarios, integrating GitOps concepts with infrastructure as code tools within automated pipelines provides a substantial chance to improve deployment efficiency and reliability. With the assistance of such concepts and tools, organizations that automate

software delivery, infrastructure management and deployment operations while maintaining version-controlled consistency can increase delivery cycles, reduce errors, and strengthen system resilience throughout the process.

This chapter provides a high-level overview of the identified problem and the need for GitOps. Together with the research question and a summary of the methodologies, it presents the purpose of this thesis project. It also includes a summary of ethics, sustainability, limitations, and the report's structure.

1.1 Background

Adopting efficient software development methodologies is significant in ensuring the timely delivery of high-quality software products. Organizations must adhere to ideal software development practices to stay at the forefront of the fast-paced software development landscape. Traditional approaches, such as the waterfall model, have given rise to agile and iterative software development methods, prioritizing collaboration and adaptability and providing a quick response to change control. Agile software development methods, such as Scrum and Extreme Programming (XP), support iterative software development cycles that provide close collaboration with multiple stakeholders, working together to deliver incremental value to the product and the company [1].

Likewise, the rise of CI/CD practices has transformed software development processes. CI/CD pipelines automate the code changes, test, build, package, and enable different stakeholders and teams to collaborate. These practices promote reliable and efficient software development. By automating repetitive tasks and facilitating feedback, these CI/CD pipelines promote agility, accelerate time-to-market, secure deployments, and enhance the overall quality of the end product. Moreover, the combination of developing and managing the frameworks and applications, called DevOps, has significantly impacted how the software is developed, deployed, and managed. DevOps embodies a cultural and organizational philosophy focusing on collaboration, automating tasks, and dividing the responsibility between development and operations teams. By accelerating a culture of continuous improvement and promoting communication, DevOps enables organizations to streamline processes, expedite delivery cycles, and enhance the resilience of their systems [2].

GitOps is an extension within the domain of DevOps, wherein infrastructure as code within the context of CI/CD pipelines are two emerging concepts

that have gained significant attention in the software development industry. Infrastructure as Code involves managing the Infrastructure resources through code, which enables the automation of infrastructure provisioning, planning, deploying, and configuring. This approach has enhanced scalability, repeatability, and consistency in infrastructure deployments. GitOps is a framework that extends the principles of DevOps to the management of infrastructure and applications by implementing Git as the foundation and a single source of truth. With the help of version-controlled configurations and automated workflows, GitOps simplifies the management of complex resource deployment while keeping the software integration and delivery in sync with the help of CI/CD pipelines. This promotes collaboration and enhances traceability to a great extent.

1.2 Problem

The goal of both practices—DevOps and GitOps is to streamline software delivery and infrastructure management processes. Although the approach varies from both perspectives, each showcases its own set of challenges. It is essential for organizations to understand these differences while seeking to adopt and inherit the methodologies into their software development and infrastructure deployment practices.

One of the major challenges in adopting DevOps lies in fostering collaboration and driving culture transformation between both the development and the operations teams [3]. DevOps utilizes shared responsibility, accountability, and a culture of continuous improvement, which requires organizations to overcome cultural resistance and facilitate cross-functional collaboration. The problem lies in the final stage of the pipelines, which is the continuous delivery and deployment pipelines adhering to DevOps methodologies, which cannot automate the infrastructure deployment, especially in keeping the desired infrastructure manifest and existing infrastructure resources in sync. In this case, infrastructure engineers are usually involved in manually applying the infrastructure changes in order to sync the infrastructure and the final state of the pipeline. There is also a lack of state management of the infrastructure resources, which causes a massive gap between the desired and the current state of the resources. Therefore, every time a declarative infrastructure deployment is passed through the pipeline, an administrator needs to manually intervene to run an imperative command to pass the infrastructure deployment. This process breaks the flow of automated end-to-end software and infrastructure delivery [4].

GitOps, on the other hand, introduces a shift in how infrastructure and application deployments are managed and synced. GitOps places greater emphasis on the version control system with the help of tools such as Git. The declarative infrastructure source code, along with the application source code, are all placed in Git repositories. GitOps is a framework and an extension of DevOps that uses Git as the single source of truth and utilizes GitOps methodologies, such as pull-based methods with a set of tools, to keep the declarative infrastructure in sync within the context of the desired and current state [5].

Therefore, to understand the importance of GitOps and to comprehend the integration of GitOps practices with CI/CD and Infrastructure as Code, the main research question to answer is:

- How does implementing GitOps and Infrastructure as Code principles impact deployment efficiency, reliability, and security in real-world software development environments?

The research question is focused on the deployment efficiency, reliability, and best practices of security for a framework built in the context of CI/CD pipelines using platforms and tools such as GitLab, Amazon Web Services (AWS), GitOps operator, and Infrastructure as Code tools. The framework involves a comprehensive development of a CI/CD pipeline that automates the application integration, test, build, push, deployment of the infrastructure resource, and deployment of the application using a GitOps operator.

A comprehensive and detailed structure of the pipeline and the architecture is explained in Chapter 3. The research question can be split into the below sub-questions (SQ):

- SQ1: How does adopting Infrastructure as Code tools affect the efficiency and reliability of infrastructure deployment processes?
- SQ2: To what extent are GitOps methods, such as push and pull methods, efficient in software and infrastructure modifications?
- SQ3: What are the best security practices in GitOps, and which tools can be employed to integrate security in the CI/CD pipelines?

1.3 Purpose

The purpose of this research is to understand how GitOps, an extension of DevOps, seamlessly applies the principles of DevOps in automating

infrastructure deployment and provides empirical evidence on how the principles of GitOps and various Infrastructure as Code tools within the context of CI/CD pipelines impact deployment efficiency and reliability. This contributes to advancing knowledge and offering practical insights for improving software development practices, especially for organizations that are considering or undergoing digital transformation initiatives, particularly in CI/CD pipeline automation and infrastructure management. In addition, the research helps researchers who are interested in expanding their understanding of DevOps, software engineering, and automation, as well as practitioners in the software development sector, by providing them with insights into new trends and best practices for streamlining deployment operations.

1.4 Ethics and Sustainability

The thesis project focuses on GitOps using GitLab in the context of CI/CD pipelines on the AWS cloud platform. As the research is comprehensive, the technical implementation would require a consistent deployment and deletion of cloud resources. This deployment directly impacts the data centers in which the resources are deployed. Although 90% of the electricity used by Amazon is sourced from renewable sources, the creation of resources will impact the energy, cooling, and power consumption in the data centers [6]. Therefore, sustainability plays a significant role in this thesis project, and significant steps have been implemented to reduce the impact and ensure the thesis project adheres to sustainability.

Additionally, the thesis project has been performed in collaboration with Scania CV AB. Therefore, the following are the factors that address ethics and sustainability for the thesis project.

1. Data Privacy: Several measures have been taken to maintain integrity, safeguard data, and ensure the privacy and security of any sensitive data collected during the research process.

2. Transparency: Potential conflicts of interest or biases were disclosed transparently to maintain the integrity of the thesis project.

3. Environmental Impact: Efforts were made to minimize the environmental impact of the thesis project, such as reducing energy consumption during data collection and analysis. For instance, regularly destroying

the resources that are being deployed into the AWS cloud platform for research.

1.5 Research Methodology

A comparative analysis using both qualitative and quantitative approaches is used in this thesis project. A framework is developed, which includes a full-stack Node.js application in a controlled GitOps environment. CI/CD pipelines are developed within GitLab and applied to automate the process of building, testing, and deploying the application as a containerized image. A GitOps operator and a set of infrastructure as code tools are utilized to deploy the infrastructure and host the application in the AWS cloud platform, and a comparative study of the infrastructure as code tools is carried out based on parameters such as deployment efficiency and reliability. Furthermore, using a GitOps operator, a comparison of push and pull methods is carried out. A detailed architecture of this framework, along with the pipeline flow, has been explained in Chapter 3. The process involves the following steps:

1. Literature Review: The academic research and industry reports on GitOps, IaC, and CI/CD practices are thoroughly reviewed to understand current knowledge.

2. Implementation and Testing: The controlled test environment includes a framework with a comprehensive Development, Security and Operations (DevSecOps) pipeline. This environment automates the code integration and testing stages, builds and pushes the application as a containerized image, and hosts the application in AWS through infrastructure deployment and GitOps operator. Tools like GitLab and Terraform, Pulumi, Kaniko, Helm, SonarQube, Trivy, Open Worldwide Application Security Project (OWASP), Argo CD, AWS and Amazon Elastic Kubernetes Service (EKS) are utilized to simulate real-world scenarios.

3. Data Collection and Analysis Guidelines: As described in Chapter 4, Sub-questions 1 and 2 contain 3 scenarios each. To answer sub-question 1, data is collected for 200 pipelines based on each designed scenario to compare Infrastructure as Code tools within the context of automated pipelines. To answer sub-question 2, the data collection is performed 30 times based on each designed scenario to compare GitOps methods within the context of automated pipelines. To answer sub-question 3,

the best practices of security in GitOps are reviewed, and a set of tools are experimented with the implemented framework to secure the CI/CD pipeline. These tools are described in Section 2.3.

Since the sample sizes in sub-questions 1 and 2 are always greater than 30, the Central Limit Theorem (CLT) can be applied. This theorem states that the distribution of a sample variable can be approximated to a normal distribution (i.e., a "bell curve") if the sample size becomes larger (n $\geq$ 30) regardless of the population's actual distribution shape [7].

When two independent populations (µ1, µ2) are compared, the difference in the population means is the parameter of interest, i.e., µ1 - µ2 [8]. Samples from two populations are independent if one has no connection with the other [8].

In this thesis, the comparison of IaC tools and GitOps methods in sub-questions 1 and 2 is between two independent populations. Therefore, the point estimate for the difference in population means is the difference in sample means [8]:

$$(\bar{X}_1 - \bar{X}_2)$$

Where $\bar{X}_1$ and $\bar{X}_2$ denote the mean of the first and second samples.

As a result, to answer both sub-questions, Confidence Intervals (CI) are needed to quantify the uncertainty in the point estimate with a certain level of confidence (usually 95%).

To calculate the Confidence Interval (CI), the pooled estimate S_p of the common standard deviation is calculated [9]:

$$S_p = \sqrt{\frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}}$$

Where S_1 and S_2 are the sample standard deviation of the first group and second group, and n_1 and n_2 are the sample sizes of the first and second group.

Finally, the Confidence Intervals (CI) for a difference between two means, when both n_1 and n_2 are greater than 30, is given by [9] [10]:

$$C.I = (\bar{X}_1 - \bar{X}_2) \pm Z S_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

Where the Z-Value is ≈ 1.96 for 95% confidence.

All the above formulas are valid when the two populations have similar variances. As a guideline, the ratio of the sample variances, $\frac{s_1^2}{s_2^2}$, must be between 0.5 and 2.

4. Recommendations Formulation: Based on the findings, practical recommendations are provided for organizations adopting GitOps principles and IaC practices, along with formulating the best practices of security in GitOps with the experimentation of security tools. These recommendations may assist organizations in adopting GitOps and improving infrastructure and application deployment processes and the CI/CD pipeline security.

The results and their analysis are described in detail in Chapter 4.

1.6 Delimitations

The thesis project examines solely the effects of GitOps and Infrastructure as Code on deployment efficiency, reliability, and security, deliberately excluding consideration of other aspects within the field of software development. The various excluded aspects include the application's programming language, User Experience (UX), project management tools, application architecture such as microservices and monolithic and software development practices such as agile and scrum. This deliberate focus allows for a more in-depth analysis of the impact of GitOps and IaC on deployment processes, ensuring a thorough understanding of their effects in these key areas. However, this narrow scope may overlook other potentially valuable insights that could contribute to a broader understanding of software and infrastructure development practices.

Furthermore, with limitations on time and resources, there may be constraints on the length of the observation period, potentially restricting the ability to fully capture the long-term effects of implementing GitOps and IaC on deployment processes and organizational dynamics. Despite efforts to mitigate these constraints, such limitations could influence the comprehensiveness and depth of the findings. The author got familiarised with the tools before determining which ones to use for the technical implementation, as the author had no previous experience with some of the technologies and tools. Because of this, there is no bias towards any one set of tools or technologies, and the technical setup's approach ensures an unbiased

comparison. Additionally, there is a risk of generalization of findings, as the study's conclusions may be specific to the particular context and tools used in the research. The metrics may differ based on different public cloud vendors. While the findings may offer valuable insights within the scope of the study, their applicability to other software development environments may be limited.

1.7 Structure

Chapter 1 of the thesis outlines the primary concepts and justification for the research, as well as a brief historical overview that sets the stage for the other chapters. This chapter also explains the research methodology and approach for the sub-questions. A thorough exploration of cloud computing, DevOps, DevSecOps, Infrastructure as Code, container technologies, GitOps concepts, and technologies related to the thesis are covered in Chapter 2. It also focuses on previous research and studies based on GitOps, its concepts, and its impact on CI/CD pipelines.

Chapter 3 provides information on how technologies and tools are picked and a thorough explanation of how the selected tools, technologies and chosen platforms are implemented and utilized. It also includes a detailed explanation of how the selected tools are implemented in the controlled GitOps environment, abstracting the architecture and pipeline flow. In Chapter 4, the results and outcomes from the analysis of collected data are explained.

Chapter 5 provides an overview and summary of concepts drawn from experimental efforts and literature research. Quantitative and qualitative characteristics, like implementation efforts and the traceability of deployed infrastructure, are covered. Chapter 6 concludes the thesis by summarising its key findings and providing suggestions for future research based on limitations and areas that need more investigation.

Chapter 2

Literature Study

This chapter summarises and introduces the related research areas, platforms, and technologies that are mainly utilised in DevOps. The following categories comprise these fields of research. The related work in GitOps within the software development sectors is also covered in this chapter.

- Cloud Computing
- Introduction to DevOps
- DevSecOps and Security in DevOps pipelines
- Infrastructure as Code
- Container Technologies
- GitOps

2.1 Cloud Computing

With the introduction of cloud technologies and platforms, businesses can simplify their operations with endless cloud solutions that suit their needs. Cloud computing technologies and the concepts of DevOps are linked, as they provide a foundation for implementing DevOps practices, including GitOps, CI/CD, and IaC. Furthermore, when cloud computing and DevOps work in sync, they offer significant strengths that benefit businesses.

As a result, integrating DevOps concepts with cloud technologies assists organizations in achieving greater stability and scalability in their operations, mainly when automated pipelines are designed to perform continuous

integration, run automated tests, and deploy applications in a highly scalable cloud infrastructure resource [11].

To understand the relevance of cloud computing in DevOps, it is essential to comprehend its principles and concepts. The National Institute of Standards and Technology (NIST) provides a comprehensive definition of cloud computing, which serves as a widely accepted reference. This definition describes the characteristics, deployment, and service models of cloud computing [12].

NIST specifically outlines five characteristics, three service models, and four deployment models.

1. Characteristics: The on-demand and self-service model allows end users to provide cloud resources as needed without requiring any human intervention from the cloud service provider. This model ensures a pay-as-you-go model, meaning that the end user only pays for the time the resource has been utilized. Broad network access ensures that the provisioned cloud services are accessible over the internet. Furthermore, resource pooling involves aggregating the cloud resources to serve multiple users, providing a dynamic allocation of cloud resources. Finally, elasticity enables a flexible and cost-effective way to auto-scale up and down the resource as per the resource workload.

2. Service models: The service models include Infrastructure as a service (IaaS), Platform as a service (PaaS) and Software as a service (SaaS). Infrastructure as a service (IaaS) provides virtualized computing resources, including virtual machines, networking, and storage solutions. This service allows the users to manage the underlying infrastructure hosting the resources. PaaS offers a development platform for users with services for deploying, building, and managing applications. In this case, the complexity of infrastructure management is abstracted. Software as a service (SaaS), a service that delivers software applications, allows users to access and use applications hosted on a subscription model. In this case, there is no need to maintain or install the software.

3. Deployment models: Public, private, hybrid, and community cloud are four deployment models in cloud computing. Public clouds are owned by cloud service providers, such as AWS, Microsoft Azure, and Google Cloud Platform (GCP). Private clouds are dedicated to a single organization or a business and can be hosted on or off-premises,

providing greater control of security and customizability. Various organizations share community clouds with a common interest, such as regulatory complaints. Finally, hybrid cloud models combine both public and private clouds, allowing organizations to utilize the benefits of both deployment models.

2.1.1 Amazon Web Services

AWS is a cloud computing platform that follows a public cloud deployment model owned by Amazon. AWS provides various cloud computing services, including computing resources, storage and networking options, PaaS, and other Infrastructure as a service (IaaS) services. A report based on the cloud infrastructure market states that AWS holds a massive 31% share in the public cloud service providers [13]. To comprehend the concepts further, it is essential to understand AWS and some of its services that are widely utilized in the cloud computing architecture.

1. Virtual private cloud (VPC): A VPC lets users design their network architecture. Subnets, network gateways, routing tables, and IP address ranges are examples of VPC components. It is a crucial part of the platform and serves as a parent resource for other networking components in AWS.

2. Subnet: Subnets are child resources within a VPC, allowing users to partition the VPC network into subnets. Public and private are two types of subnets, giving direct access to the internet and providing isolation, respectively.

3. Internet and Network Address Translation (NAT) Gateway: An Internet gateway is a component within a VPC that allows the cloud resources within the same VPC to access the Internet directly. A Network Address Translation (NAT) gateway provides isolation and access to resources in a private subnet while preventing inbound internet.

4. Elastic Compute Cloud (EC2): An elastic compute service by AWS that allows users to rent virtual servers (Instances). These EC2 instances are replicas of regular host computers available in the market.

5. Availability Zones: These are locations within a specified region designed to be isolated from failures in other zones. They provide high availability and fault tolerance to ensure that applications and

infrastructure resources are always healthy and the entire system has a high uptime.

6. Auto-scaling Group: This component allows the instances to automatically scale up and down based on the custom configuration defined by the user.

7. Route 53: This service in AWS is a highly scalable Domain Name System (DNS). It allows the creation of ANAME records of a domain and provides hosted zones, domain registration, and domain routing.

2.2 Introduction to DevOps

Adopting software development best practices ensures the delivery of high-quality, reliable and scalable software products. The waterfall model, a traditional software development model, has gradually introduced agile and more iterative approaches that focus on stakeholder collaboration, adaptability and responsiveness. Agile software development practices emphasize immediate customer and stakeholder feedback and consist of frequent checkpoints in short iterations [14]. Therefore, agile teams promote flexibility and responsiveness to evolving requirements in the software development process. On the other hand, CI/CD practices have revolutionized the software development and delivery process by automating code integration, facilitating rapid software delivery, and conducting tests in several stages and deployment. This allows the teams to detect and fix the integration issues early, ensuring that updates, time to release the product into the market, and product updates are quick, focusing on improving the overall software quality [15]. The software development industry needed a revamp in its methodologies and how software is built and delivered for end users. With the evolution of DevOps practices, Infrastructure as Code, and GitOps, the industry has taken a significant leap. Table 2.1 summarizes the software development and delivery model differences [16].

Table 2.1: Difference Between Agile, DevOps, and CI/CD

Agile	**DevOps**	**CI/CD**
Iterative processes	Interconnecting tasks	Integration + Deployment
Feedback and Test cycles	Focus on Culture	Focus on automation
Software-defined life-cycle	Cross-functional roles	Highlighting tools

2.2.1 Agile Software Development

Agile software development is a traditional model with an incremental and iterative approach to software delivery. It promotes flexibility and collaboration between stakeholders and the teams and seeks feedback from them throughout the development process [17]. Scrum, Kanban, and Extreme Programming approach (XP) are such frameworks that have various common characteristics which share an agile software development model:

1. Incremental and Iterative Development: Teams that follow the agile development model work in short iterations usually sprints, lasting to a few weeks, aiming to deliver software updates and increments at the end of each iteration. This allows the teams to seek feedback frequently from the stakeholders and allows the teams to respond quickly to the dynamic requirements. Most companies today follow the agile software development model [18].

2. Collaboration: The agile method prioritizes close collaboration with stakeholders throughout the iterative development model. Continuous improvement and feedback are two significant roots of the Agile development model. The teams and developers ensure that the end-users are satisfied with the products and that they can meet their needs.

3. Dynamic Planning: The agile software development model embraces changes dynamically. The requirements may evolve with time during iterative sprints. Therefore, this model does not have a rigid or predefined plan. The planning and requirements are dynamic and adjusted based on the feedback received from the stakeholders.

4. Cross-functionality: Agile teams are usually self-organizing, which allows team members to make decisions to meet the project's goals. Team members collaborate while sharing knowledge and expertise to meet the needs of a high-quality product.

5. Continuous improvement: Agile promotes an incredible culture of continuous improvement, and teams seek feedback from each other and reflect on their processes. For example, during the regular checkpoint meetings, the teams collect the feedback received and reflect on it as part of the product development process. This continuous improvement identifies the drawbacks and makes enhancements to improve the quality of the end product.

2.2.2 History of DevOps

DevOps is a combination of both "development" and "operations." DevOps has its foundation in IT operations practices and software development. DevOps gained popularity and momentum in the early 2000s, and its origin can be traced back to various antecedents in the operations management and software development industry. The early influences of DevOps concepts are observed in the Agile Manifesto 2001, which supported collaborative and iterative approaches to software development [14].

The growing culture of DevOps as a distinct discipline in the software industry can be linked to crucial events in the mid-2000s. Patrick Debois, a software engineer from Belgium, organized the first "DevOpsDays" conference in 2009, where he introduced the term "Agile Infrastructure" to describe the alignment of operations, development and practices. At the same time, engineers at Flickr presented their experiences achieving quick deployment cycles, with up to ten deployments per day, showcasing the benefits of continuous delivery and automation [19]. The term "DevOps" was coined by Patrick Debois during discussions at the Agile 2008 conference, marking the formal inception of the DevOps culture. Patrick used the term to encapsulate the cultural and technical collaboration between development and operations teams, emphasizing shared goals, values, and practices, which today is commonly referred to as DevOps culture[20].

Organizations used various tools and techniques to automate infrastructure provisioning in public clouds, configuration management, and deployment pipelines as DevOps principles and practices became more formalized over the 2010s. During this time, the DevOps community expanded quickly, creating online forums, conferences, and physical meetup sessions devoted to exchanging best practices and case studies. With surveys and publications showcasing the financial value of DevOps practices—such as faster time-to-market, increased software quality, and greater organizational agility—the industry's acknowledgement of DevOps as a strategic imperative also grew. DevOps reflected a broader cultural shift towards collaboration and automation across organizations, extending beyond IT operations to security, quality assurance, and business operations.

DevOps continued to develop throughout the 2020s, paralleling developments in containerization, cloud-native technologies, and orchestration tools like Kubernetes. Emerging concepts like "DevOps 2.0" addressed

the difficulties of managing large, distributed systems and microservices architectures by emphasizing observability, "shift-left" testing, and site reliability engineering (SRE) [21].

2.2.3 DevOps Best Practices

"The Phoenix Project" by Gene Kim, Kevin Behr, and George Spafford introduced the concept of the "Three Ways," which serve as fundamental principles underlying DevOps practices [22]. The following principles tend to foster collaboration, communication, and cooperation among all stakeholders involved in software development to achieve continuous delivery and organizational improvement [23].

1. Workflow Principle: The workflow principle in the DevOps culture focuses on streamlining and automating the Software Development Life-cycle (SDLC). This enables faster delivery of high-quality software. This principle is built upon agile methodologies to simplify and enhance the workflow from the stages of the development environment to the production environment. Therefore, DevOps promotes cross-functional collaboration by breaking the hierarchy between multiple teams and stakeholders [24].

   Automation plays an important role in implementing the workflow principle, as it enables the automation of repetitive tasks, such as code integration, various types of testing, delivery and deployment, leading to improved efficiency, reduced cycle times, and increased productivity. Through adopting Continuous Integration (CI) and Continuous Delivery and Deployment (CD) practices, organizations can achieve a seamless and reliable flow of changes from development to production, facilitating rapid iteration and feedback loops.

2. Feedback Principle: The feedback principle highlights the significance of acting upon feedback received throughout the software delivery process. This principle is rooted in the Agile principles of iterative development and continuous improvement. Organizations can identify issues at an early stage, gather insights, and make informed decisions to enhance software quality and user satisfaction by regularly seeking stakeholder feedback, including customers, end users, and internal teams. Feedback loops are integral to DevOps practices such as continuous monitoring, automated testing, and user-centric design [24].

3. Experimenting and continuous learning: This principle emphasizes a culture of innovation, experimenting, and continuous learning. This principle is a best practice and encourages organizations to inherit a culture where employees are encouraged to take risks, experiment, and learn from failures and successes. In simple terms, it is a blame-free work culture.

2.2.4 Lifecycle of DevOps

The concept of the DevOps lifecycle is often depicted as a continuous process, reflecting the iterative and collaborative nature of DevOps practices. DevOps practitioners commonly represent this lifecycle using an infinite loop or circular model to illustrate how the various stages of the DevOps journey are interconnected and interdependent. While these stages may appear sequential, the loop symbolizes the fundamental principle of continuous collaboration and iterative improvements supporting the DevOps philosophy [25].

This approach emphasizes integrating development, operations, and cross-functional teams throughout the software delivery lifecycle, fostering shared responsibility, and typically involves planning, development, testing, deployment, monitoring, and feedback. However, unlike traditional waterfall methodologies, where each step follows a linear progression, the DevOps lifecycle promotes continuous feedback loops and constant iteration [26].

Additionally, ongoing communication and collaboration between development, operations, and other stakeholders facilitate identifying opportunities to improve software product processes and innovate. Organizations acknowledge software development and operations iterative and cyclical nature by visualizing the DevOps lifecycle as a continuous loop. This model emphasizes the importance of constant collaboration, feedback, and improvement across all lifecycle stages, enabling organizations to deliver high-quality software efficiently, adapt to changing requirements, and drive continuous innovation and value delivery within the software product.

2.2.5 DevOps Processes, Technologies, and Tools

In DevOps, processes, technologies, and tools integrate closely to facilitate the seamless integration of software development and IT operations. This integration enables organizations to achieve faster delivery cycles, higher-quality software, and improved team collaboration.

The DevOps Process
DevOps processes revolve around automation, collaboration, and continuous improvement. Continuous Integration (CI) and Continuous Delivery and Deployment (CD) are two core processes in DevOps. Continuous integration involves developers frequently integrating code changes into a shared repository. Each integration triggers automated builds and tests, ensuring that the codebase always remains functional and deployable.

Continuous delivery and deployment extend Continuous Integration by automating the deployment of code changes to staging and production environments after successful testing. Continuous Delivery and Deployment (CD) pipelines facilitate the rapid and reliable release of software updates, promoting shorter feedback loops and faster time-to-market, which means faster software delivery. Unlike continuous delivery, where changes are automatically deployed to staging or pre-production environments for further validation before manual approval for production deployment, continuous deployment automates the direct deployment of code changes to production environments after successful testing. Additionally, DevOps emphasizes practices such as Infrastructure as Code, which enables the automation of infrastructure provisioning and adopting Agile methodologies to promote iterative development.

Technologies and Tools:
Various technologies support DevOps practices, helping teams automate processes, manage infrastructure, and streamline collaboration. VCS like Git enables teams to manage and track code-based changes, facilitating versioning effectively and rolling back to previous versions without hassle, providing flawless traceability. Collaboration extends to tools like Slack and Microsoft Teams to facilitate communication among team members, promoting transparency. CI/CD platforms such as Jenkins, GitLab CI/CD, and CircleCI automate the build, test, and deployment processes, allowing for continuous integration, deployment, and delivery of software changes. Monitoring and observability tools like Prometheus, Grafana, and Datadog provide insights into pipeline performance and infrastructure health, enabling teams to detect and address issues proactively [27].

It is essential to understand the Git terminology to understand the concepts further. The GIT workflow is illustrated in Figure 2.1. Git revolves mainly around repositories, which are directories that hold project files and historical metadata. Repositories can be hosted remotely on platforms such as GitHub

and GitLab or locally. The working directory, staging area, local repository, and remote repository are the four primary regions developers interact with within a Git repository. Developers modify project files in the working directory. The index, commonly called the staging area, is an intermediate phase before committing changes to the repository. The whole history of commits and changes made to the project on the developer's computer is stored in the local repository. The remote repository acts as a shared location for team members and mirrors the project's history. Each modification in the project is recorded as a commit with a message, an immediate representation of the change at that specific moment. Git branches enable parallel development by creating unique paths for new features without affecting the main codebase. The project's stable version is contained in the master or main branches. A developer can push changes from a local repository to a remote repository using git push. Pull merges changes from a remote repository so that the local repository is always up to date. Furthermore, checkout is necessary to transition between branches, and merging merges the modifications from different branches into a single branch.

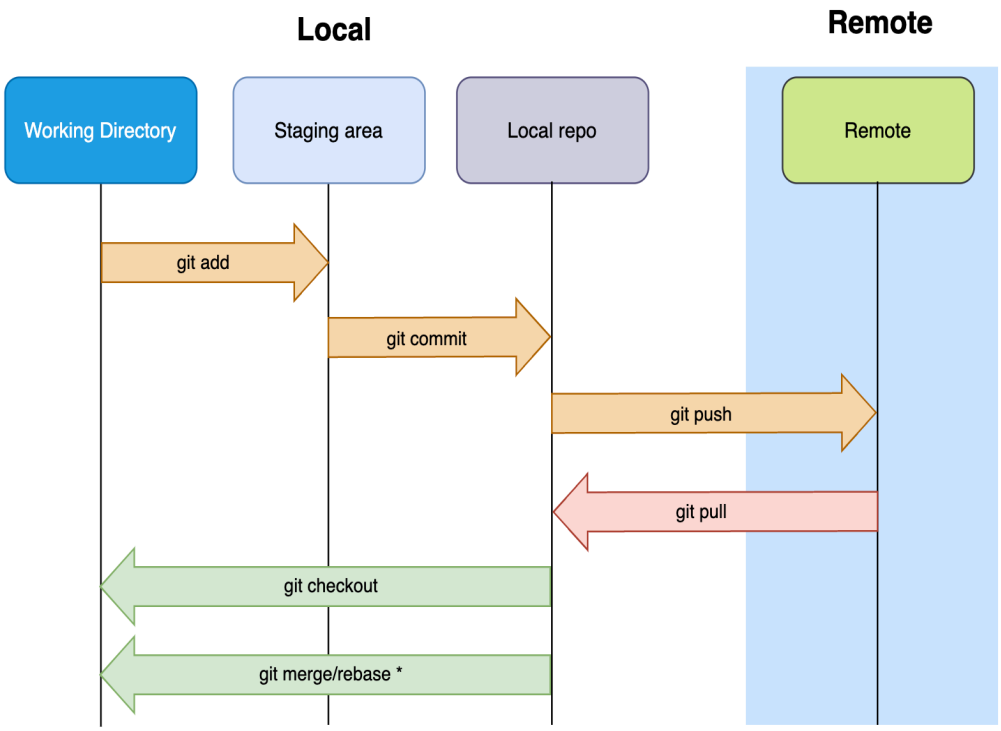

Figure 2.1: Git Workflow

Ultimately, DevOps processes, technologies, and tools work in sync to enable organizations to achieve their goals of delivering high-quality software rapidly and reliably. By embracing automation, collaboration, and continuous improvement, DevOps empowers teams to innovate and adapt quickly to technological advancements.

2.2.6 DevOps Pipelines

Software development, testing, and deployment are orchestrated by a set of automated steps called a DevOps pipeline. The code flows smoothly from one step to the next in a streamlined procedure that ends with the end users. Pipelines are the foundation of DevOps practices, enabling development and operations teams to work together productively and produce high-quality software rapidly and frequently.

As illustrated in Figure 2.2, a DevOps pipeline typically consists of several interconnected stages or components, each performing specific tasks in the software delivery lifecycle. These specific stages are orchestrated using specialized DevOps platforms, such as Jenkins, GitLab CI/CD, or Azure DevOps, which automate the execution of tasks and manage the flow of changes through the pipeline. By automating repetitive tasks and streamlining the software delivery process, these pipelines enable teams to release the end software product more frequently, with iterations, while maintaining higher quality and reducing manual effort and errors. These stages may include:

1. Version Control: Application developers commit code changes to a version control system (e.g. Git). This ensures that all code modifications are tracked and easily managed.

2. Continuous Integration (CI): In this stage, code changes are automatically built, tested, and integrated into a shared repository.

3. Automated Testing: Various automated tests (e.g., unit tests, integration tests, regression tests) are executed to validate the application's functionality, performance, and security. Automated testing ensures consistent and reliable results while reducing manual efforts.

4. Continuous Delivery: Continuous Delivery involves deploying code changes to testing or staging environments after successful integration and testing. In this case, the decision to deploy to the production environment is a manual approach.

5. Continuous Deployment: Once changes have been thoroughly tested in the staging environment, they can be automatically deployed to production environments. Deployment automation ensures consistency and reduces the risk of human errors during deployment. This is where the concept of GitOps operators makes an impact.

6. Monitoring and Feedback: Continuous monitoring of application performance, infrastructure health, and user feedback provides valuable insights into the application's behavior. Monitoring helps detect issues early and enables operations teams to respond quickly to incidents. Tools like Datadog, Prometheus, and Grafana are a few examples. They offer a rich visualization platform and a front-end user interface, allowing users to create customizable dashboards displaying metrics and logs from various sources. AWS CloudWatch also provides a comprehensive monitoring and observability service tailored to AWS cloud resources and applications.

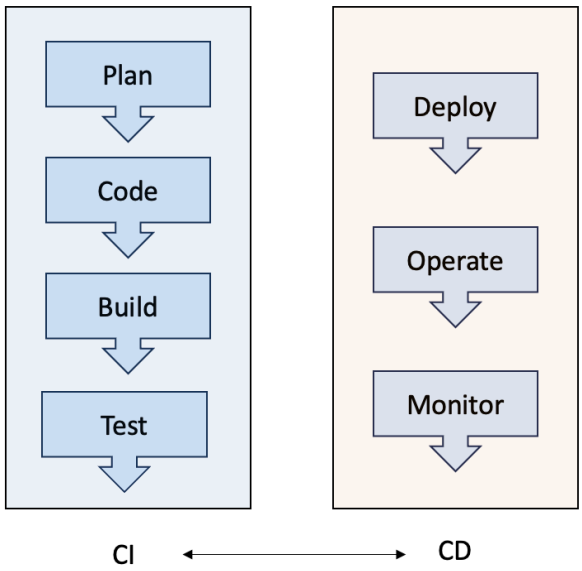

Figure 2.2: A Typical DevOps Pipeline

2.2.7 Agile Methodologies with the DevOps Approach

Software development approaches have evolved with the use of agile methods when combined with the DevOps methodology. Their goal is to increase the

seamless integration of efficiency, automation, and collaboration throughout the software delivery lifecycle. Since automation is a fundamental DevOps concept, it promotes agility and leads to a smooth, end-to-end DevOps pipeline.

The combination of these two models has revolutionized software delivery and collaboration in today's Information Technology (IT) company operations. The primary component is CI/CD, which ensures that code changes are promptly tested and implemented. This implies that teams can quickly add new features and fix bugs. Additionally, because it enables teams to use code to automate the setup and management of the infrastructure, IaC is an excellent fit with Agile. This maintains uniformity and aids in keeping various teams informed. DevOps places a strong emphasis on constant monitoring and feedback. It indicates that teams can identify bugs at an early stage, address them quickly, and improve software over time.

As a result, combining Agile with DevOps fosters a culture where teams collaborate closely, exchange ideas, and support one another. Teams may receive feedback during sprints, automate repetitive processes, and continuously improve software using the combination of Agile and DevOps.

2.3 DevSecOps and Security in Pipelines

DevSecOps practices incorporate security into the DevOps methodology. Integrating security testing, vulnerability assessment, and compliance checks at every stage of the development process seeks to guarantee the security of software development pipelines. By addressing security issues early in the pipeline, vulnerabilities can be prevented from reaching production systems. Vulnerabilities in pipelines constructed using improper methods pose serious risks. These flaws can potentially cause data breaches, illegal access, and other security incidents that compromise sensitive data availability, confidentiality, and integrity. For instance, an attacker could insert malicious code into software through a pipeline vulnerability, spreading to users and eventually causing damage or financial loss.

As per research conducted by the National Institute of Standards and Technology (NIST), pipelines built with inadequate standards may be susceptible to many security risks [28], such as:

1. Vulnerabilities in pipelines setup: Vulnerabilities due to improper setup

of pipeline configurations might result in misconfiguration and possible security risks, and attackers may take advantage of weak credentials.

2. Vulnerabilities for malware: Vulnerable software components can be an entry point for introducing malware.

In this case, a fully developed application can have several weak points within the filesystem of the source code files, vulnerable modules and packages, and the versions of every component being utilized in the code, which may cause a significant risk before they are utilized in the production environments. Therefore, ensuring the security of the stages within a pipeline, containerized applications, and the source code is critical.

2.3.1 Tools in DevSecOps

This is where tools like OWASP, SonarQube, Trivy, and Kubescape shine, offering robust security checks to safeguard against vulnerabilities throughout the development lifecycle.

- **OWASP (Open Web Application Security Project)**

  Offering an abundance of tools for web-based application security, the Open Web Application Security Project (OWASP) is a nonprofit organization. Their project, API Security, aims to secure APIs, which are being used more and more for integrating web apps with external systems. To assist security experts in securing APIs and web applications, OWASP offers a variety of tools [29]. Developers can prioritise security efforts by using its well-known OWASP Top Ten, which lists the most important security threats that web applications have to deal with [30] [31]. Developers may perform dynamic security testing and find vulnerabilities like weak authentication and exposed sensitive data with the use of tools like ZAP (Zed Attack Proxy).

- **SonarQube**

  SonarQube is an open-source tool created by SonarSource for automated code reviews and static code analysis to find errors and bugs in 29 different programming languages. SonarQube analysis is set up at the start of the pipeline for a thorough scan of the application code [32]. As a result, SonarQube helps developers identify and fix security vulnerabilities, code smells, and other issues in their overall codebase.

SonarQube integrates with various platforms, including Jenkins and GitLab, to provide a comprehensive security solution.

- **Trivy**

  Trivy is a lightweight, open-source vulnerability scanner explicitly designed for containers and other CI/CD pipeline artifacts. It excels in identifying security vulnerabilities within container images and dependencies. Trivy supports various container image formats and package managers, leveraging vulnerability databases like Common Vulnerabilities and Exposures (CVE) to deliver accurate and up-to-date vulnerability information [33]. By seamlessly integrating Trivy into continuous integration pipelines, vulnerability scanning is automated as part of the build process, ensuring that only secure container images are deployed to production environments. In this thesis project, Trivy performs a file system scan on the application directory and scans the container image upon every build and push [34].

- **Kubescape**

  Malicious users may always exploit security by violating Kubernetes security best practices. As a result, mitigation techniques must be proposed to protect the cluster [35]. Kubescape is an open-source Kubernetes (K8s) security platform that integrates seamlessly with CI/CD pipelines and K8s clusters. Kubescape fits seamlessly into the DevSecOps landscape, including security complaints and misconfiguration scanning parameters. Kubescape offers features like a Command Line Interface (CLI) and multiple output formats. Utilizing Kubescape allows administrators to seamlessly secure and gain insights into the security perspective of K8s clusters. Currently, it is a Cloud Native Computing Foundation project (CNCF) [36].

By incorporating Kubescape, SonarQube, OWASP, and Trivy into continuous integration pipelines, organizations may establish a robust security layer that addresses diverse security concerns throughout the software development lifecycle. Together, SonarQube, OWASP, and Trivy assist development teams in confidently building and deploying secure applications, mitigating security risks, and ensuring the integrity of their software assets.

2.4 Infrastructure as Code

Infrastructure as Code is a fundamental practice in modern DevOps methodologies, revolutionizing how cloud infrastructure is managed and provisioned in public cloud platforms. Additionally, the market size of the IaC globally is expected to reach 2.8 billion by the year 2028 [37]. IaC involves defining infrastructure resources and configurations through machine-readable definition files rather than relying on manual processes or interactive configuration tools. This approach aims to automate and standardize infrastructure provisioning and management for enhanced efficiency and reliability [38] [39].

2.4.1 IaC Concepts, Benefits, and Challenges

- **Key Concepts and Principles of IaC:** IaC allows infrastructure to be defined either declaratively, describing the desired state of the infrastructure, or imperatively, specifying step-by-step instructions to provision the infrastructure. Furthermore, IaC templates should exhibit idempotency, ensuring that applying the same configuration multiple times results in the exact desired state, regardless of the initial infrastructure state. Versioning is crucial in managing IaC templates, enabling teams to track changes and roll back to previous versions when necessary [39].

- **Benefits of IaC:** Automation lies at the core of IaC, facilitating the automation of infrastructure provisioning, configuration, and management tasks. This reduces manual effort and errors and improves consistency across infrastructure deployments. Moreover, IaC enables scalability, allowing organizations to scale infrastructure resources dynamically in response to changing demands. Re-usability is another significant advantage, as IaC templates can be reused across different environments, promoting consistency and reducing duplication. Finally, IaC enhances agility and speed in the deployment process [40].

- **Challenges:** Adopting IaC requires a shift in mindset and skill set for infrastructure and operations teams, who may need to learn new tools, languages, and best practices. Managing complex infrastructure configurations and dependencies in code can pose challenges, necessitating careful planning, modularization, and abstraction. Additionally,

ensuring the security and compliance of infrastructure configurations in IaC templates requires thorough testing, auditing, and validation processes.

- **Declarative and Imperative syntax:** These are two methods to achieve the desired state of the infrastructure. In the case of the declarative, the focus is on specifying in extreme detail what should be achieved without detailing how to achieve it. Therefore, this involves describing the desired outcome of the infrastructure, and the entire system figures out the steps to achieve the declared syntax. Meanwhile, imperative syntax emphasizes specifying steps and commands to achieve the desired state. It usually involves instructing the system on how to act in sequence. Yet another markup language (YAML) syntax is one such example of a declarative syntax. The declarative syntax is usually associated with high-level abstractions, whereas the imperative syntax is procedural and often used in lower-level programming languages. Table 2.2 summarizes the differences in declarative and imperative programming, which is essential to comprehend the concepts further.

Table 2.2: Differences Between Declarative and Imperative Approaches in IaC

Declarative syntax	**Imperative syntax**
Specifies the desired state of the infrastructure without defining the step-by-step process to achieve it.	Defines the step-by-step process to achieve the desired state of the infrastructure.
Focuses on the outcome rather than the procedure.	Focuses on the procedure rather than the outcome.
Utilizes tools like Terraform or Kubernetes YAML files.	Utilizes configuration management tools like Chef, Ansible or Puppet
Changes are made by updating the desired state, and the tool determines the steps to achieve the state.	Changes are made directly through commands or scripts that modify the current state of the infrastructure.
Easier to understand and manage, especially for large and complex systems.	Can lead to more complex and error-prone configurations as the infrastructure scales.

2.4.2 Infrastructure as Code Tools

Table 2.3 summarizes the differences in the Infrastructure as Code tools [41] [42]. Additionally, the following are the major Infrastructure as Code tools that are being utilized in the technological sector:

- **Terraform, and Pulumi**

  In the domain of Infrastructure Provisioning Tools, Terraform stands out as a versatile and powerful solution. Developed by HashiCorp, Terraform enables users to define infrastructure resources across multiple cloud providers and on-premises environments using HashiCorp Configuration Language (HCL). Terraform's state management capabilities ensure safe and predictable provisioning of resources, making it a preferred choice for managing infrastructure as code in dynamic and multi-cloud environments [43].

  With Pulumi, another IaC tool, developers can define and manage cloud infrastructure using well-known programming languages like Python, JavaScript, TypeScript, and Go. Pulumi allows the developer to use code to define cloud resources. Pulumi was made open-source in 2018 and was launched by Joe Duffy, Eric Rudder, and Luke Hoban [44] [45]. This makes Pulumi the latest Infrastructure as a Code tool in the market.

- **Ansible, Chef and Puppet**

  Among the prominent Configuration Management Tools are Ansible, Chef, and Puppet. While these tools also incorporate Infrastructure as Code into their functionality, they are not primarily utilized for infrastructure provisioning. Ansible is a free and open-source automation tool that prioritises usability and simplicity. It does not require installing agents on managed nodes and communicates with servers via Secure Shell (SSH).

  In contrast, Chef is a robust automation platform that employs a master-agent design. It enables defining infrastructure as code by utilising cookbooks, which are Ruby-based scripts. Similarly, another configuration management solution based on the master agent is Puppet. It makes it possible to define infrastructure configurations in declarative Puppet language [43]. Both Chef and Puppet utilize Domain Specific Language (DSL) for their configuration code.

Table 2.3: Comparison of Configuration and Infrastructure as Code Tools

Tool	Type	Language	Agent-based
Terraform	Declarative	HCL	-
Ansible	Both	YAML	-
Chef	Imperative	Ruby-based DSL	✓
Puppet	Declarative	Puppet DSL	✓
Pulumi	Declarative	Multi-support	-

2.4.3 State Files, Current and Desired State

State files are essential to IaC and are used to manage the infrastructure. These files record the desired and current state of the infrastructure components. The desired state describes the configuration the system wants to achieve, whereas the current state shows how the infrastructure is configured. It is a reference point, helping tools determine what adjustments are necessary to bring the desired and current states into sync. The provisioning, configuration, and management of resources can be automated using the IaC tool by comparing the current and desired states specified in these files [46].

A successful IaC deployment requires precise state file maintenance. This reduces the possibility of configuration drift and inconsistencies by ensuring that infrastructure updates are appropriately tracked. State files also make the history of infrastructure updates visible, which makes auditing easier. Furthermore, managing and storing the state files safely is critical for teams to work in sync. Usually, these state files are stored in a central location, such as S3 Bucket in AWS, a cloud object storage service provided by Amazon. State locking at the time of infrastructure provisioning is important so that no two developers can utilize the same state file, reducing the risks of conflicts. Therefore, only one developer can update the state file at a time. One such solution is the DynamoDB table in AWS [47]. Terraform state file management is depicted in Figure 2.3, with a central state file location in an Amazon S3 bucket. This state file only permits one developer or a system to update, provision, and delete the resource without causing conflicts because it is locked using the AWS DynamoDB table.

Figure 2.3: AWS State Management for Terraform

As illustrated in Figure 2.4, the flow represents the importance of the state files in infrastructure as code. A developer commits the declarative infrastructure code using a version control system, and the system eventually runs the pipeline to provision the infrastructure. Initially, when the declared tool invokes the IaC syntax, the tool creates a state file stored either locally or in a cloud object storage such as S3 and updates the state file to match the current state of the infrastructure. If the desired and current states are not in sync, the tool either performs a rollback or updates the infrastructure. The goal is always to match the declared and the desired state of the infrastructure.

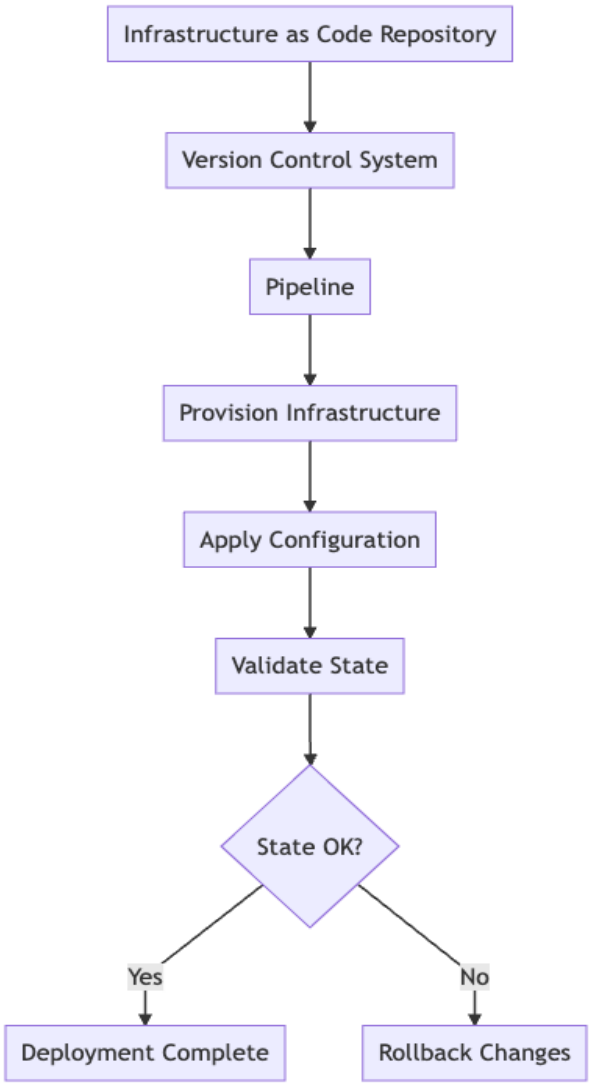

Figure 2.4: Importance of State Files in Infrastructure as Code

2.5 Container Technologies

Container technologies represent an essential shift in modern IT practices and industries that fundamentally represent how the application is developed, built, stored, and managed. Container technologies originated from the early concepts of FreeBSD Jails and Solaris Zones [48]. The fundamental principles include a portable and lightweight nature, which enables consistent execution of the application across diverse environments. Consider an application with several different modules, libraries, and dependencies; Because of dependencies and occasionally even hardware configuration, the application structure cannot ensure that this application will operate successfully on all host machines. All things considered, sharing an application between host machines is time-consuming and challenging.

In this case, container technologies bundle the entire application as an image into a small, lightweight instance that may be stored in a hub or central repository. If developers can access the hub, they can run the container image on the host computer without installing any dependencies or libraries.

Leading tools and technologies in the container ecosystem include Docker, Kaniko for building docker images, Kubernetes as a container orchestrator, Helm as a package manager for Kubernetes and container runtimes like Containerd, CRI-O, and CI/CD pipelines for automated testing and deployment.

2.5.1 Tools and Platforms

- **Docker**

  Docker is an essential application development platform consisting of a set of services and products. Docker uses Operating System (OS)-level virtualization to package the application dependencies and all their libraries into lightweight, portable containers that host the packaged application image. A high-level docker architecture is illustrated in Figure 2.5. These docker containers have their own networking rules and isolation, providing excellent scalability. Due to their lightweight tendency, containers are resource-efficient, simplifying deployment and management while promoting consistency [49].

  In DevOps, developers can integrate Docker technologies to build, package, and push their applications to a central hub or repository. Docker images are built using a Dockerfile that describes the commands in imperative syntax.

Docker Hub is a native container image platform that hosts publicly available images. Therefore, anyone with docker installed on their host machines can efficiently run an application image as a container.

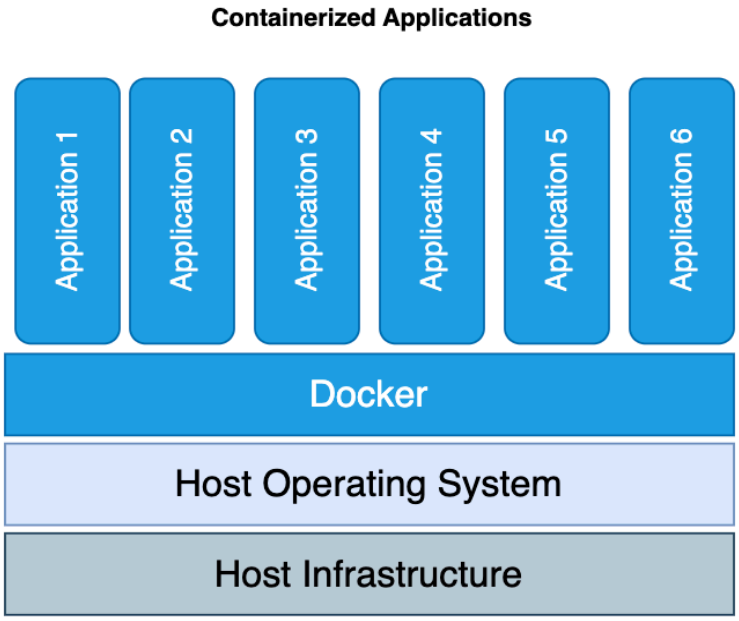

Figure 2.5: Docker Architecture

- **Kaniko**

  Kaniko, an open-source project by Google, creates Docker images inside the containers without the intervention of Docker. This revolutionizes the process of creating container images. Kaniko has gained a lot of traction because it can run in an unprivileged container, which provides greater security and portability, which is essential in K8s. The Cloud Native Computing Foundation (CNCF) has observed Kaniko's distinct methodology, enabling developers to build images inside K8s clusters, improving workflow efficiency, and facilitating CI/CD pipelines efficiently. Kaniko also has a caching feature that enhances the speed of building the Dockerfile.

- **Kubernetes**

  K8s, an open-source container orchestration platform, is primarily known as Kubernetes. K8s orchestrates the containers built by

containerization platforms such as Docker. It also abstracts complexities and container deployments in the K8s clusters [50]. Applications running on K8s automatically scale, self-heal, and incorporate load balancing and service discovery. K8s may be easily integrated with various technologies and is available in declarative and imperative syntax. Kubernetes (or) K8s manifest files are declarative syntax and deployment files written in YAML [51]. Additionally, major cloud providers such as Google Cloud, AWS, and Microsoft Azure also offer PaaS K8s services in their respective cloud platforms.

To comprehend the concept of K8s, it is essential to understand the following associated components [52] [53], and the Figure 2.6 illustrates the Kubernetes architecture, which is adapted from [54] [55].

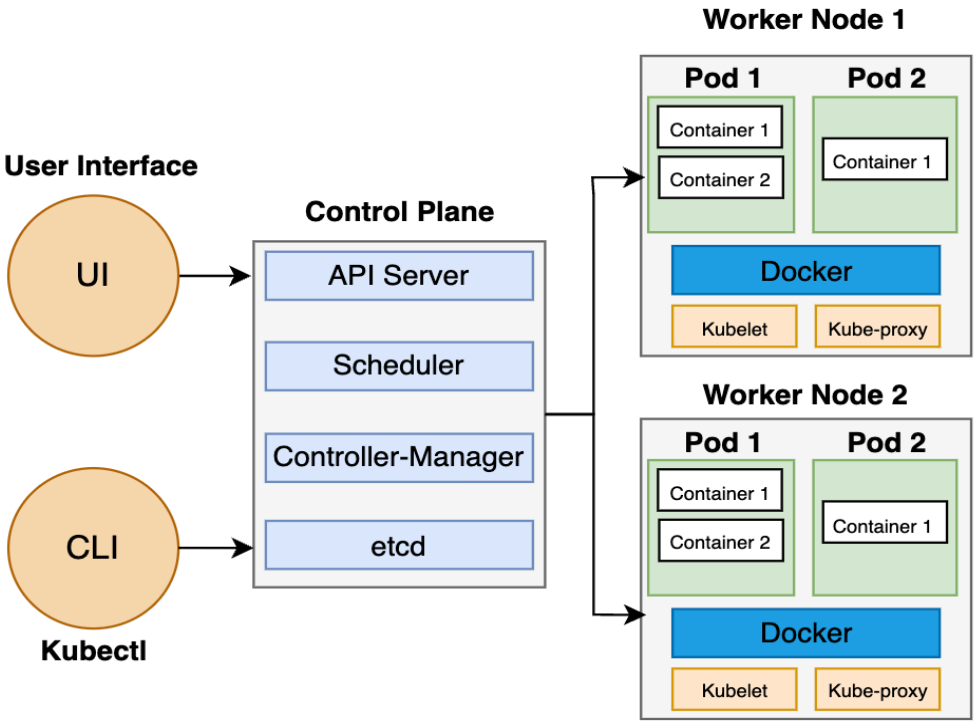

Figure 2.6: Kubernetes Architecture

2.5.2 Kubernetes Concepts

- **Kubernetes cluster:** A K8s cluster combines workloads, containers, manifests, deployments, and all essential resources required to keep the cluster running. A K8s cluster can combine multiple master and worker

nodes based on the utilized architecture. For example, public cloud platforms have one master node for each cluster, which the platform manages as they are PaaS services.

- **Master nodes:** K8s master nodes are the brain and are responsible for managing the entire control plane of the cluster. The master node hosts critical components such as the API server, controller manager, scheduler, and etcd. Firstly, the API server is the front end for the cluster, which exposes the API and serves as an entry point for administrators to interact with. The controller manager ensures that the cluster's desired state matches the actual state by continuously reconciling the resources. The scheduler is an essential component for assigning workloads to specific worker nodes based on resource availability. Lastly, the etcd is a key-value store distributed in nature that stores the cluster configuration and state information. Furthermore, kubectl is a Command Line Interface (CLI) tool for interacting with the clusters. As a result, master nodes coordinate and orchestrate the cluster operations, ensuring scalability, fault tolerance, and high availability.

- **Worker nodes:** Worker nodes, also known as minions, are responsible for running the workloads and containers in the cluster. They host the application containers and allocate the necessary computing resources to run them. Kubelet runs on each worker node and communicates with the master node's API server. These also run components such as Kube-proxy, which is responsible for networking within the workload and the master node. Docker and containerd are examples of container runtimes that run the container.

- **Kubernetes pods:** A pod is the smallest and most fundamental unit of deployment. A pod represents a single instance of a running process within the cluster. Pods can contain one or more containers that share storage and networking configuration resources. As part of best practice, it is recommended to have one container per pod [56].

- **Kubernetes services:** A service is an abstraction that defines a logical set of pods and policies to access them. ClusterIP and NodePort are examples of the K8s service, which exposes the service on an internal IP address and a static port on each node's IP address, respectively. The former service allows external traffic.

- **Kubernetes ReplicaSet:** ReplicaSet is a type of controller that ensures

a specific number of identical pod replicas are running simultaneously. It allows high availability of the hosted application, supports horizontal and vertical scaling, and provides rolling updates to ensure seamless application uptime.

- **Kubernetes deployments and namespaces:** Deployments in K8s are resource objects that abstract the deployments and scaling of pods. Deployments provide declarative updates to replica sets and pods, ensuring that the desired state in the configuration is always maintained. Namespaces provide a way to partition and isolate the cluster's resources logically. Namespaces serve as a virtual cluster, and each namespace can hold multiple deployments, and each deployment can hold an abstraction of services, pods, replica sets, and application deployments. Multiple namespaces help developers to handle their own set of deployments separately.

2.5.3 Helm

When an application is deployed into K8s clusters, it consists of various declarative manifest files that are complex to manage. When this is considered on a large scale, the orchestration and management of the manifest files are very complex to handle. This is where package managers such as Helm play an essential role.

Helm works on the K8s platform and is used to create application packages that make deploying and managing applications easier. Helm consolidates the K8s application into the helm chart package, making it easier to update, install and manage the applications. These charts come in handy when deploying intricate applications on hybrid or multi-cloud infrastructures because all the K8s manifests and dependencies the application requires are consolidated into charts. As a result, Helm chart provides the best practices for managing complex applications running on Kubernetes while assisting the developers with deployment cycles. Besides, Helm has rollback and version control functionality that allows developers to roll back to a working state in case of an interruption. Thus, with the help of Helm, Kubernetes application deployment becomes more efficient in terms of productivity and, at the same time, reduces time to delivery, as it provides a clear and well-structured procedure for managing apps in Kubernetes clusters [57].

2.6 GitOps

GitOps is an extension of the DevOps methodology. The GitOps concept was popularized by Weaveworks, which specialized in Kubernetes and cloud-native technologies. The term GitOps was coined by the engineers of Weaveworks in 2017 and has gained enormous traction [58]. The company Weaveworks has recently announced that it is shutting down [59].

GitOps is a modern software development and operations model focusing on leveraging Git as the foundation and a single source of truth for maintaining and defining the infrastructure and application code. GitOps extends the IaC principles by incorporating the version control system into the process, like Git, into the continuous delivery and deployment pipelines. This method emphasizes saving the application and infrastructure deployment codes in two different Git repositories and also gives a single and traceable environment for managing the changes in the infrastructure and the application. These are incorporated into the DevOps culture with the help of pipelines, which in turn makes it an even more automated software and infrastructure deployment process.

2.6.1 GitOps Architecture

GitOps focuses on continuous infrastructure delivery and deployment. Therefore, the infrastructure codebase is written in a declarative format, and version control systems such as Git serve as the central control plane for managing the entire infrastructure and software delivery lifecycle. Any codebase changes and infrastructure configuration are proposed via Pull (PR) or Merge Requests (MR) to the Git repository.

Once a pull request is merged into the repository's main branch, usually a GitOps operator, implemented as a controlling running within the Kubernetes cluster, detects the changes and applies them automatically to the target environment within a specific time. There are two different deployment methods in GitOps: pull and push-based deployment. This is explained in the following subsections. As a result, this overall process ensures that the desired state, declared in the Git repository, is always in sync with the current state of the infrastructure. This process is illustrated in Figure 2.7.

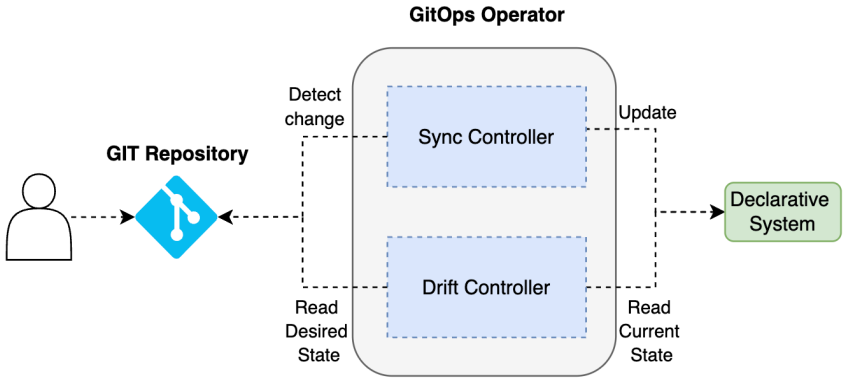

Figure 2.7: High-level GitOps Architecture using GitOps Operator

2.6.2 GitOps Operators

GitOps operators are the primary controllers responsible for implementing GitOps principles. Currently, they are utilized only within a Kubernetes environment. As a result, these operators automate the deployment, monitoring, and management of the deployed applications by continuously syncing the desired state of the system with the current state of the system. The two most well-known GitOps operators are Flux and Argo CD. In May 2024, Argo CD had nearly 16,000 stars, whereas Flux had about 6,000 stars on their GitHub repository [60] [61].

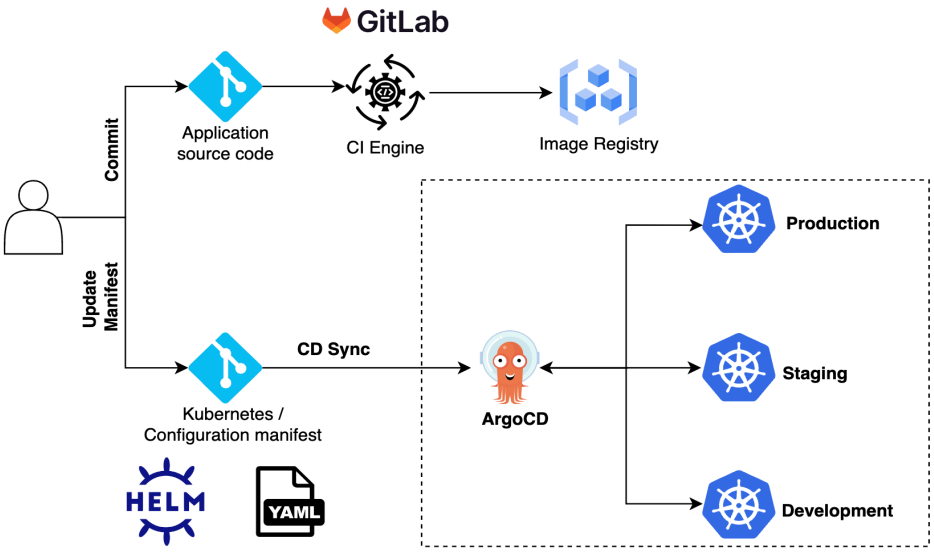

Figure 2.8: Pull-based GitOps Architecture using Argo CD

Figure 2.8 illustrates the GitOps architecture using Argo CD. Additionally, Argo CD always requires a commit in the application repository for the changes to be detected, and it is possible to set an application reconciliation timeout of 1 second in Argo CD, which allows the operator to pick changes within a second of a git commit.

2.6.3 Principles and Advantages of GitOps

GitOps primarily has four major principles, which are evident in the official practical guide to GitOps by Weaveworks [62]. These specify the core principles that GitOps inherits to differentiate itself from various software and infrastructure delivery methodologies. These principles are the following:

- **The entire system is declarative:** This implies that the infrastructure resources are always defined in a declarative format. For example, K8s can be defined by a set of facts instead of instructions in a declarative format. With the configuration being declarative, it is possible to roll back to previous states quickly and quickly redeploy it in case of disaster recovery.
- **Git as the single source of truth:** Git is an essential version control system that is the foundation of infrastructure repositories. The declarative configuration stored in Git implies that there is only a single point from which the entire system is derived. This creates massive support in the case of rollbacks to a previous version. As a result, this principle states that the entire system must be version-controlled.
- **GitOps operator:** This principle states that if the changes are approved and merged into the main or master branch, they are automatically applied to the system. The significance of this principle is that access to the infrastructure hosting the resource or the K8s cluster is optional.
- **Ensure that the present state is accurate:** This principle highlights that the GitOps operator should notify the developer if there is a discrepancy between the current state and the desired state of the system, as long as the state is defined in a declarative format and is version-controlled. Because this principle improves the system's ability to repair itself, an operator can always alert the developer when the infrastructure is out of sync and allow it to heal itself in case of a human error. In this case, the system is managed continuously by the GitOps operator, which also provides frequent feedback.

2.6.4 Pull vs Push in GitOps

GitOps utilizes two methods to deploy changes from the desired state to the system. These methods are called “Push” and “Pull”, respectively. A push-based method is invoked when the changes are pushed to Git repositories or when the changes are merged into the main branch, which serves as a trigger. The trigger performs and deploys the changes to the system. This method fosters a continuous delivery approach, facilitating a seamless declarative change transition into the system.

Therefore, by automating the deployment processes based on the events of version control, the Push-based method ensures that the declared desired state of the system, defined in Git, is consistently reflected in the system as part of Pull and merge requests. It is important to note that there is no intervention from a GitOps operator in this case.

In the pull-based method, a dedicated GitOps operator is utilized, which is configured into the cluster beforehand and actively monitors changes within the designated Git repositories. When the modifications within the repositories are detected, the GitOps operator pulls the changes from the repository and applies them to the resource, which is a K8s cluster in this case. This method ensures that the cluster’s configuration and the current state of the resource always remain synchronized with the desired state defined in the Git project repository.

This approach is known as continuous deployment, which pushes the changes into the live environment without manual intervention. With the help of this approach, organizations can ensure tighter control over the configuration changes, enabling efficient and reliable infrastructure and application management. The pull and push-based GitOps workflow is illustrated in Figure 2.9. As a result, this approach emphasizes self-healing, and automatic sync, promoting consistency across the entire system [5].

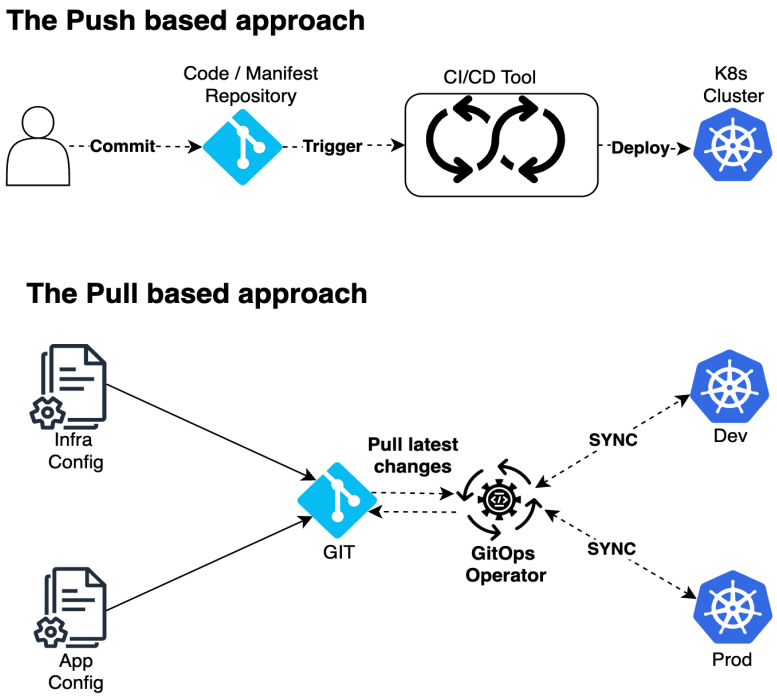

Figure 2.9: Pull and Push-based GitOps Workflow

2.6.5 Why is GitOps Necessary?

Over the past few years, the software development landscape has experienced rapid expansion, prompting enterprises to transition from outdated legacy infrastructure to modern management platforms. The reliance on manual processes within legacy systems often resulted in significant delays in the Software development life-cycle (SDLC), causing infrastructure deployment and management issues. Therefore, to tackle these challenges and roadblocks, teams began automating standard processes and configurations by creating programmatic scripts in languages such as Bash, PowerShell, or Python. Usually known as scripting languages. However, it became evident that not all issues could be effectively addressed solely through scripting.

Subsequently, configuration management tools emerged, leveraging a code-based approach to configure and manage infrastructure. This shift facilitated easier management and operation of application infrastructure at scale, reducing operational pressure. However, certain aspects, such as infrastructure provisioning in the cloud platforms, remained either manual or partially automated. In the next phase of infrastructure automation evolution, development teams are poised to focus on writing infrastructure as

code and streamlining processes further. Adopting CI/CD pipelines brought changes where hundreds of code changes were processed daily. However, this frequency of updates also introduced challenges, including the need for frequent infrastructure and configuration updates, leading to inconsistencies and prolonged troubleshooting efforts.

In response, DevOps teams turned to GitOps as a solution to standardize infrastructure and application deployment. GitOps offers a framework that can be applied to any infrastructure managed declaratively, providing a streamlined approach to managing infrastructure and configurations. Overall, GitOps represents a significant advancement in addressing the complexities of modern software development, offering standardized configuration management and deployment processes across diverse infrastructures [58].

2.6.6 Security in GitOps

Usually, in the traditional approach, the CI pipeline pushes and deploys the images from the registry to the K8s cluster. In this case, the pipeline needs access to the API credentials with the tool utilized for the CI pipeline. Furthermore, this implies that the CI tool becomes a resource with a high-value risk. It is possible to access the production environment if an attacker breaks into the CI system. This is regardless of whether the cluster is secure or not.

When GitOps methodology is utilized for infrastructure and application deployment, a GitOps operator acts on behalf of a developer, performing changes to the infrastructure. Therefore, GitOps separates the CI process from the CD, which makes it a very secure method of deploying the applications to the Kubernetes cluster. Table 2.4 illustrates how GitOps separates read and write privileges between the Kubernetes cluster, container repository, and CI/CD system.

Table 2.4: Separation of Read and Write (R/W) Privileges using GitOps

S No	CI System (Test-Build-Publish)	CD System (Sync app repo to cluster)
1	Runs outside the cluster	Runs inside the cluster
2	Read access to the code repo	R/W access to the infra repo
3	R/W access to container repo	Read access to image repo
4	R/W access to the CI environment	R/W access to the cluster

However, when the infrastructure goes down and needs instant rebuilding, restoring the previous state of the infrastructure is impossible because the CI

system doesn't have its state management in place. Therefore, it would be necessary to rerun the entire CI pipeline and jobs to rebuild everything from scratch.

2.6.7 Steps to Implement GitOps

There are four main stages to the GitOps process:

1. Commit Code Changes: Developers initiate the development of artifacts by committing code changes to the Git repository. Usually, every build and push workflow entails constructing and producing a new image. Container registries, like Docker Hub or GitLab container registry, are where the images are kept [63].

2. Deploy Application Configurations: The application's configuration settings are implemented. This means specifying deployment and related changes in YAML-formatted manifest files within the Kubernetes context. In the end, these files define the desired state of the application by stating how to map ports, services, environment variables, config maps, secrets, and configuration settings for the image. The Helm package manager may also be utilized instead of hardcoding the manifests.

3. Application Deployment: Every change pushed to the Git repository is observed by a GitOps operator. The operator contrasts the current state of the application with the intended state stated in the repository. It automatically implements the required adjustments to bring the infrastructure into compliance if any disparities are discovered.

4. Image deployment: Once the application is deployed into the cluster and the image is configured and aligned to the intended state, it is deployed to various environments, such as production and staging. The images may be pulled based on the different tags and versions in this case.

2.7 Related Work

In the field of GitOps, a limited amount of research has been conducted to thoroughly study the impact of IaC tools and their effect on efficiency and reliability. GitOps has been recently released, and as an extension to the DevOps methodology, it has gained much traction in industries where infrastructure deployment and management are crucial. Therefore, many internet blogs, YouTube videos, Medium articles, websites, and various opinions are available explaining the concept of GitOps, its terminology, relevant tools, and different deployment techniques. Sources related to Managing Applications on Kubernetes and implementation of CI/CD in GitOps were studied, and experiments were designed and conducted during the thesis implementation [64] [65] [66] [67]. Additionally, there are conference papers on analysing declarative and pull-based Deployment Models on GitOps Using Argo CD and the Implementation of GitOps in containerized infrastructure [5] [68]. The purpose of referring to these sources is to understand the kind of research conducted in the fields related to GitOps.

2.7.1 Implementation of CI/CD Pipelines using GitOps

CI/CD implementation is the major research area in this thesis, and major sources related to implementing CI/CD pipelines using the GitOps methodology are discussed in [64] [65]. A few conclusions from the existing work are significant for understanding the findings in this field. The best practices for building a pipeline that deploys applications into the Kubernetes cluster are discussed in [64], and it states specifically that the Helm chart for application deployment should be included and the GitOps methodology should be utilized. It also states that GitOps increases security and emphasizes the importance of state management in reverting to a previous state at any point. Additionally, the findings from [65] state that to increase the flexibility of the development process, GitOps methodology must be utilized, and the benefits of implementing CI/CD pipelines with GitOps concepts were identified. The below conclusions from [65] are significant to understanding how CI/CD best practices facilitate better software development.

1. Establishing a pipeline using GitLab automates Continuous Integration and Delivery/Deployment, facilitating fast integration, testing, and deployment of changes.

2. Asynchronous deployment using GitOps enhances application deploy-

ment speed and scalability, supporting parallel deployment of multiple applications.

3. GitOps principles streamline software delivery via Git, integrating development and IT operations. Infrastructure as code and continuous testing improve code quality and development efficiency.

4. GitLab supports multi-cloud environments while facilitating flexibility and improving infrastructure management.

5. GitOps provides options, rollback, and redeployment using version control systems such as Git.

2.7.2 CI/CD Tools in GitOps

The selection of selection CI/CD tools in GitOps is a significant area that could provide a preliminary idea of the already existing research and the availability of the best tool supporting GitOps in the market. The source [67] has researched CI/CD tools in GitOps and concluded that Argo CD showed superior performance in deployment times and recovery from faults compared to other open-source continuous delivery tools such as Drone and GoCD. This thesis also stated how the pull-based deployment model is more secure than the traditional push-based deployment models. Additionally, the source [5] has conducted a study on the rollback process in implementing GitOps using Argo CD and the importance of integrating Argo CD with other automation platforms. The study also demonstrates how pull-based and declarative deployment approaches, instead of push-based models, can enhance CI/CD pipelines.

2.7.3 GitOps in Containerized Architecture

The source [68] demonstrates a step-by-step process and implementation of GitOps in the containerized infrastructure using the K8s cluster, utilizing GitLab CI/CD, and deploying applications using Argo CD. This implementation successfully automates the deployment of application and infrastructure management, providing benefits such as scalability and reduced manual errors. As a result, this study concludes that GitOps proves effective for Containerized Infrastructure, offering automation and efficiency benefits.

2.7.4 Conclusion

As a result, from the provided thesis and research articles, the research in the field of GitOps is relatively limited. The existing research and study emphasize the importance of GitOps in enhancing CI/CD pipelines to streamline software delivery. Additionally, when integrated with Argo CD, CI/CD tools such as GitLab are identified as crucial components of GitOps implementations, with Argo CD showing exceptional performance in deployment times and fault recovery. Additionally, GitOps proves effective in the containerized infrastructure. GitOps offers a significant advantage in modern software development practices, but further research is needed to explore its full potential.

In conclusion, this thesis adds significant value to the current research and study by exploring the impact of IaC tools on deployment efficiency, reliability, and best security practices in the context of CI/CD pipelines using GitOps.

This chapter contributed a thorough literature study that explained the related research areas, platforms, tools, and technologies mainly used in DevOps. It aims to build a foundation of important concepts that are essential to comprehending the following chapters, including the design of technical setup and architecture of AWS resources.

Chapter 3

Method and Implementation

Based on the extent of existing research and experiments described in Section 2.7, the concept of GitOps, within the context of CI/CD pipelines, comparison of GitOps methods and Infrastructure as Code, is not a highly researched area. Currently, there are no articles or research related to the deployment efficiency and reliability of infrastructure as code tools or the best practices for adopting GitOps methods within the context of CI/CD pipelines at the time of writing the thesis. The methods and scenarios currently available on the internet focus on building the GitOps methodology on the local machine without the intervention of CI/CD tools. When CI/CD tools are considered in GitOps, the workflow and architecture complexity level increases; therefore, multiple tools and technologies are considered when designing such scenarios and implementing a controlled GitOps environment. The selected tools and the controlled GitOps architecture mimic a production end-to-end DevSecOps CI/CD pipeline framework.

This chapter and the sections below discuss the technologies, platforms, and tools selected for implementing a controlled GitOps environment. This chapter also discusses why some common IaC tools used in Infrastructure deployments have not been picked for GitOps implementation. This chapter follows the following structure.

- Selection of tools and technologies
- Data collection setup
- Application details
- Framework architecture and setup
- Entry point of the framework

- Pipeline workflow
- Kubernetes cluster details
- CI pipeline
- CD pipeline
- Application deployment using Helm chart
- Justifications for not selecting other tools

3.1 Selection of Tools and Technologies

A controlled GitOps setup is required as part of the technical implementation of the thesis work. Therefore, it is essential to comprehend the need for multiple tools and platforms. Furthermore, GitLab is the foundation of GitOps in this research. As a result, the entire architecture and workflow will be implemented on top of GitLab, a prominent DevSecOps platform.

The comparison of Infrastructure as Code tools in a CI/CD environment will be based on the comparison of tools Terraform and Pulumi with respect to the framework established. The implemented framework is explained in Section 3.4.

Additionally, to describe the best practices for GitOps in a CI/CD environment, a comparison of pull-and-push-based GitOps methods is measured. This also means comparing GitOps methods with and without a GitOps operator.

Finally, open-source security tools, as described in Section 2.3, are implemented in the framework to formulate the best practices of security in a CI/CD pipeline.

A GitLab runner is needed for the pipeline jobs to run in GitLab. Two Ubuntu-based EC2 instances are self-hosted on AWS and utilized as the GitLab runner for Terraform and Pulumi repositories separately. This instance is also called a build server. This build server is consistent for this thesis work and is of the AWS instance type “t3.xlarge”. This instance type offers 4 vCPUs (Virtual Central Processing Units), 16 GB (Gigabyte) of memory and 5 Gigabit network performance. Additionally, the runner cache is cleared each time a scenario of a sub-question is measured.

Table 3.1 is the most suitable technology and platform stack to conduct the designed scenarios and build the framework.

Table 3.1: Platform and Technology Stack for the Research

S. No	Process - DevSecOps + GitOps	Tool
1	CI/CD and DevSecOps platform	GitLab
2	Public cloud platform	AWS
3	Application language	Node.js
4	Infrastructure resource	Kubernetes Cluster v1.29
5	Build container images	Kaniko
6	K8s Package manager	Helm
7	GitOps operator	Argo CD
8	Infrastructure as Code tool	Terraform
9	Infrastructure as Code tool	Pulumi
10	Static Code Analysis	SonarQube
11	File and image scanning	Trivy
12	Kubernetes security platform	Kubescape
13	Application dependency check	OWASP
14	SaaS monitoring solution	Datadog
15	Open-Source monitoring	Grafana and Prometheus

3.2 Data Collection Setup

A set of tools, as mentioned in rows 14, 15 and 16 in Table 3.1, are utilized in this thesis for data collection. Datadog, a SaaS-based monitoring solution, is utilized to visualize the pipeline success and failure rates on a high level for the CI framework pipeline and the IaC CD pipelines. GitLab supports native Datadog integration, and as a result, the repositories are integrated with Datadog. Secondly, to compare the GitOps methods, which are push and pull, a Prometheus operator is installed using Helm chart in the Kubernetes cluster. Prometheus is an open-source tool that collects metrics, including time-series data, and as well as records the data with timestamps. It is essential to collect metrics such as the time taken for the K8s pods to go from waiting to running state during rollback and updates with respect to pull and push-based methods, also known as with and without GitOps operator. Prometheus also collects the time taken for both GitOps methods to notice the changes in the Helm chart. This data is visualized with the help of an open-source visualization tool called Grafana. Appendix A.1.5 is a custom Python script built to fetch the pipeline run-time metrics from GitLab using GitLab APIs. Grafana and Prometheus are port-forwarded locally and are exposed to localhost on a custom port.
The detailed results, data collection and analysis are described in Chapter 4.

3.3 Application Details

GitOps requires an application deployed only within the Kubernetes cluster. This is due to the fact that GitOps operators currently support Kubernetes as an infrastructure resource. The concept of GitOps operators, such as automating the deployment, monitoring, and managing the deployed applications by continuously syncing the system's desired state with the system's current state, will be applied based on the deployed application. The kind of application, architecture, programming language, functionality, or complexity is irrelevant since GitOps operators can apply the concepts of GitOps on any application deployed into the Kubernetes cluster. However, for the thesis, an application that manages intricate synchronization processes for web requests relying on multiple dependencies and libraries is developed.

The application is a full-stack multiplayer game server built using Node.js. It allows multiple players to play "Multiplayer-PingPong" together in real time. The application uses two main libraries: Express.js for handling web requests and Socket.IO for managing real-time communication and synchronization between players.

The code execution begins with the import of relevant modules, including Express, Socket.IO, and a unique Game class. Subsequently, it configures an Express application and designates a port for the server to listen on. When run locally on a host, this application listens on port "3000", with the HTTP module facilitating server establishment and real-time communication via Socket.IO. The code manages various user interface tasks, such as Game matchmaking, handling disconnections, setting usernames, and establishing connections to the server. Additionally, a mechanism is incorporated to enable users to start playing the game as soon as they join. The workflow of the Node.js application is depicted in Figure 3.1.

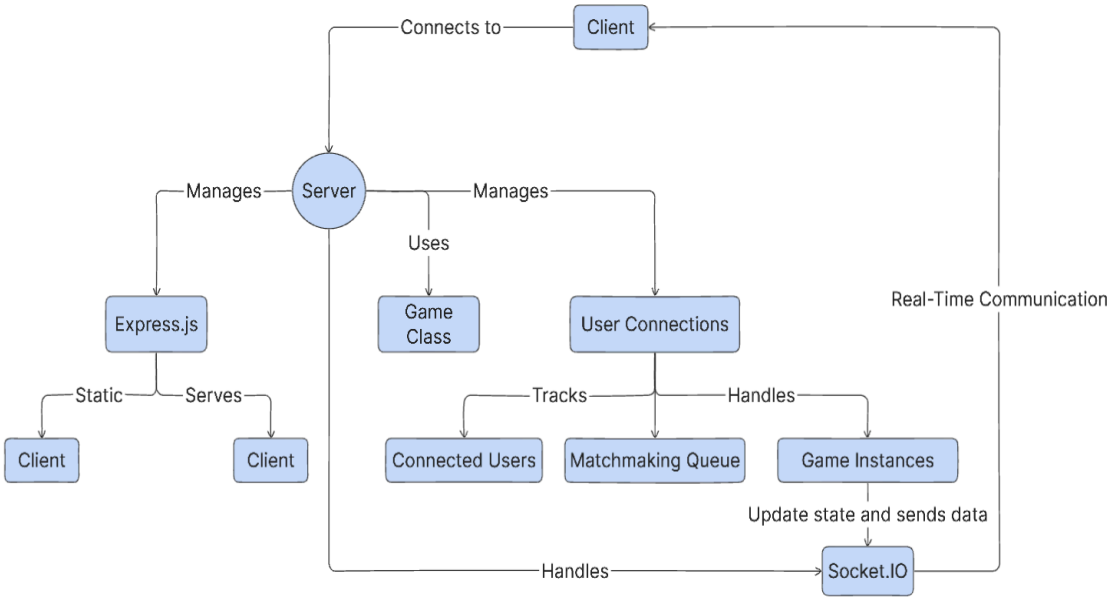

Figure 3.1: Node.js Application Workflow

During the game loop, the players receive new data from the server, which regularly updates the game state for every ongoing Game. The Game class defines the logic, including how the ball moves and interacts with player paddles and what occurs when a player gets a point. Additionally, separate Mocha unit tests are developed to ensure the application is fully functional. Mocha is a test framework based on JavaScript that runs on Node.js and facilitates asynchronous testing. Figure 3.2 and 3.3 illustrate the unit test workflow and its output.

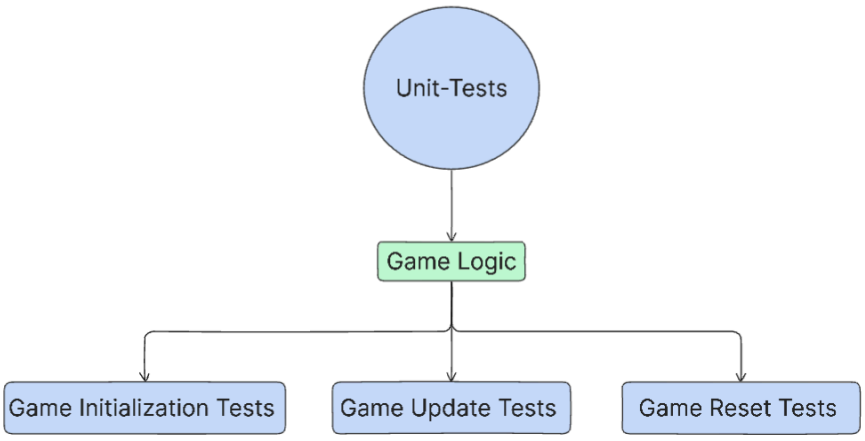

Figure 3.2: Node.js Unit-Test Workflow

```
  Ping Pong Game
    GET /
      ✔ should return the main HTML page
    Socket Connection
New client connected: Xupd9TgrZqvZVr3RAAAB
      ✔ should connect to the socket server
    Game Logic
      Game Initialization
Client Disconnected: Xupd9TgrZqvZVr3RAAAB
        ✔ should initialize a game with correct initial values
      Game Update
        ✔ should update the game state correctly after scoring
        ✔ should update ball position and velocity correctly
        ✔ should update ball position and velocity correctly when hitting players
      Game Reset
        ✔ should reset the ball position and velocity correctly for player 1
        ✔ should reset the ball position and velocity correctly for player 2
```

Figure 3.3: Node.js Unit-Test Outputs

The application's entry point has also been split into development and production environments running on two commands with node and nodemon. The latter speeds up the development of the application and automatically restarts the application when new changes are detected. This is usually a bad practice in Node.js application deployments, as downtime will occur during the process. Therefore, node <entry point/server file> is utilized in production environments. In this case, the entry point is index.js. Additionally, this application is exposed to an internet-facing application load balancer in AWS, and an ANAME record of the Fully Qualified Domain Name (FQDN) is utilized to view the application globally. The main domain is *.manish-thesis.com. Appendix A.1.4 is the source code of this full-stack Node.js application.

3.4 Framework Architecture and Setup

The framework, illustrated in Figure 3.4, represents the overall framework implemented for the research. Furthermore, every project repository established in this thesis is private. This framework consists of various pipeline stages and jobs, as explained in Section 3.6. The pre-requisite of this framework is to deploy a Helm chart in a new GitLab repository that can deploy the application based on the container registry image. The following are the various stages of the pipeline execution.

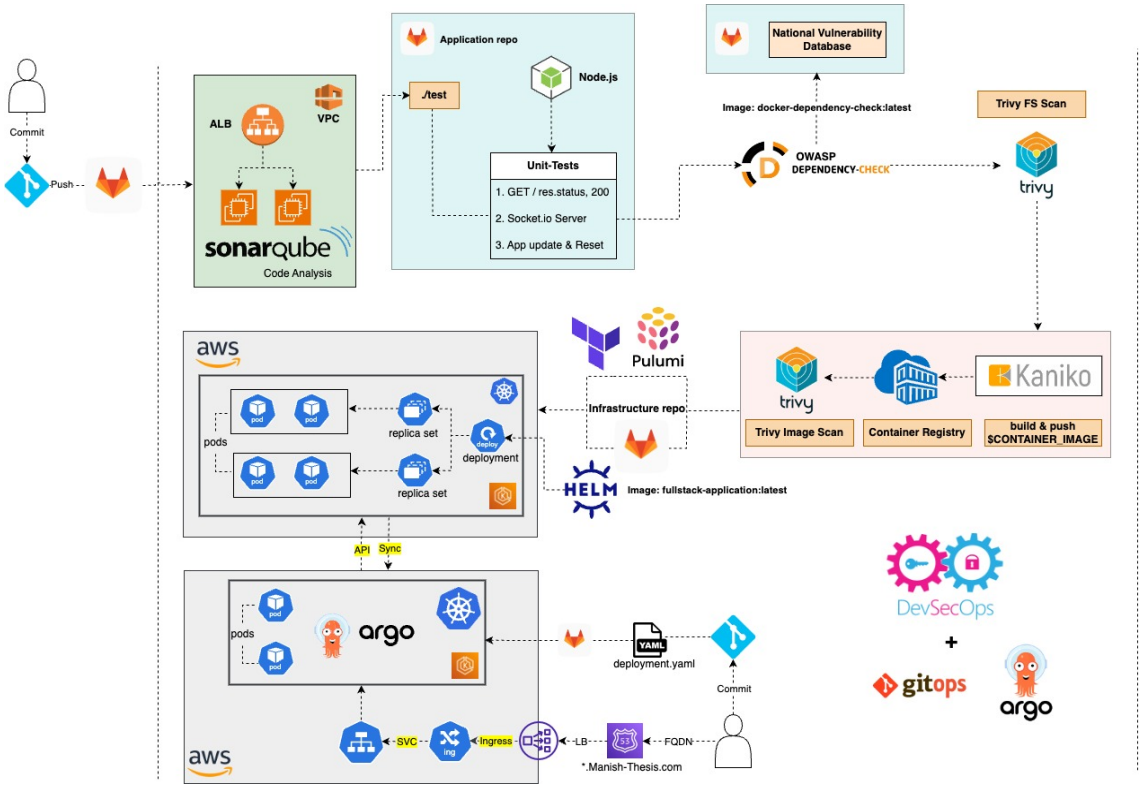

Figure 3.4: Framework Designed in GitLab CI/CD

1. The developer performs a commit to the version control system in the application repository, in this case, GitLab.

2. The first stage of the pipeline is a SonarQube code analysis that performs a static code analysis in the application's root directory. The SonarQube is hosted on two EC2 instances on AWS and is load-balanced with the help of a public Application load balancer. This allows the developer to expose the SonarQube dashboard on the internet Hypertext Transfer Protocol Secure (HTTPS) using AWS Certificate Manager (ACM) Secure Sockets Layer (SSL) certificate. SonarQube user interface can be accessed through an ANAME record hosted using Route 53 in AWS.

3. Next, the unit-testing stage is invoked, and unit-testing is performed, as explained in the previous section.

4. The next state performs a dependency check with the help of OWASP, which checks for dependencies based on the National Vulnerability Database (NVD) by NIST.

5. The next stage performs file scanning using Trivy, and then the image is built and pushed to a container registry using Kaniko. In this case, the images are stored in the GitLab container registry.

6. Once the image is pushed and stored, a final trivy image scanning is performed, and the CD pipeline, which is hosted on a different GitLab repository, is invoked, and a Kubernetes cluster is deployed as per the declared configuration. This is known as a "Multi-Project" or a "Downstream" pipeline.

7. Once the Kubernetes cluster is deployed into AWS, the stages of the pipeline include the installation of Argo CD and application deployment using a Helm chart. In this case, the application is the latest image of the application that was stored in the GitLab container registry.

The above-mentioned stages are performed with just a simple commit. This is the overall framework utilized to perform the analysis described in Section 3.1. The developer may access the Argo CD user interface using the ANAME record of a Fully qualified domain name (FQDN) and view the application. This domain name is registered in AWS as a hosted zone in Route 53, a service in AWS, and Argo CD is exposed through an internet-facing application load balancer. Additionally, as a result, once the overall pipeline is executed successfully, the below stages are executed in sequence:

1. SonarQube code analysis
2. Unit-testing
3. OWASP dependency check
4. Trivy file-scanning
5. Kaniko image build and push
6. Trivy image-scanning
7. Kubernetes cluster deployment using IaC tools
8. Installation of Argo CD, Helm and application using Helm chart
9. Exposing Argo CD in an internet-facing application load balancer.
10. Destruction of resources that were created during the pipeline.

3.5 Entry Point of the Framework

The framework consists of an entry point described in Figure 3.4. This is the SonarQube open-source community edition hosted within EC2 instances. It is important to perform static code analysis and improve the project's code quality. Since it is at the beginning of the framework, it is important to maintain a high availability of the SonarQube instance. Therefore, two EC2 instances are load-balanced using an AWS Application Load Balancer, which works on layer 7 of the open systems interconnection (OSI) model. The SonarQube web user interface is exposed using Hypertext Transfer Protocol Secure (HTTPS), and to secure the application, an AWS Web Application Firewall is deployed to secure the application load balancer. As a result, a secure SonarQube instance is exposed to the internet using appropriate firewall rules. Figure 3.5 shows the architecture for this entry point.

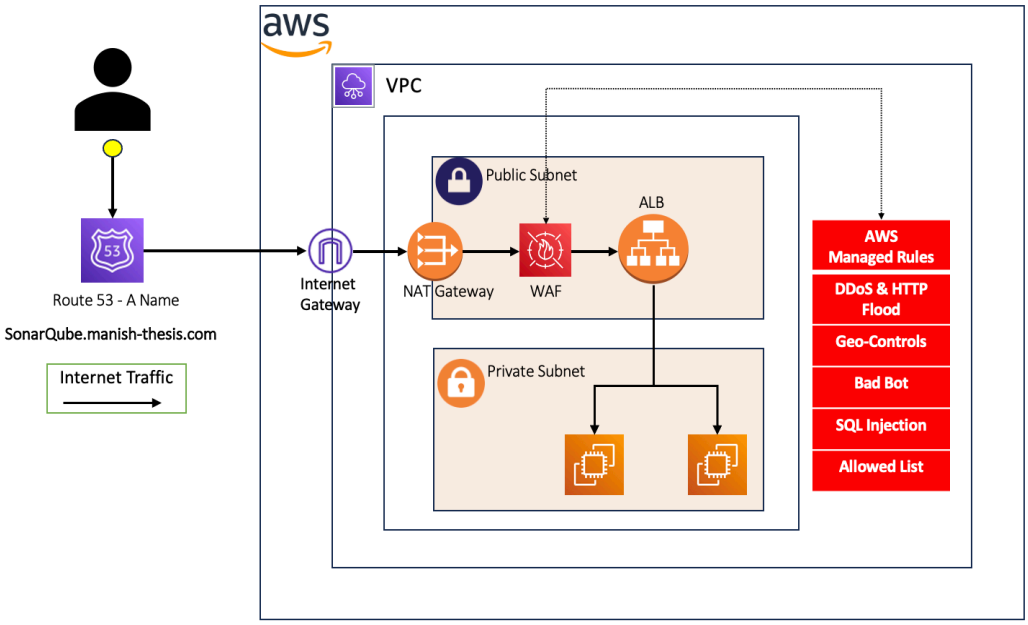

Figure 3.5: Architecture of SonarQube Instance in AWS

3.6 Pipeline Workflow

The workflow is a better way to comprehend the architecture of the framework. The following illustrations represent the pipeline stages on a high level and replicate the architecture as explained in Section 3.4. Figure 3.6 and 3.7

represent the pipeline workflow using Terraform and Pulumi, respectively.

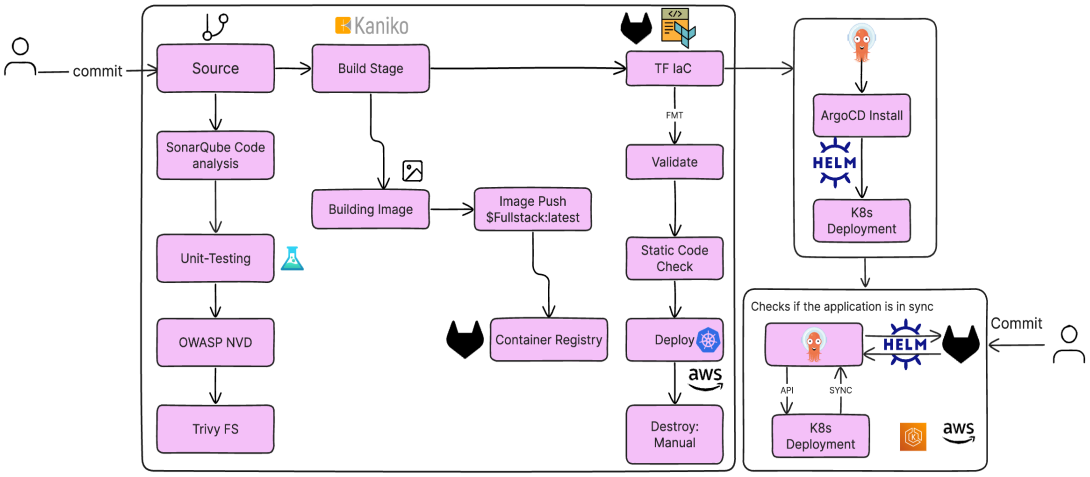

Figure 3.6: Pipeline Workflow using Terraform

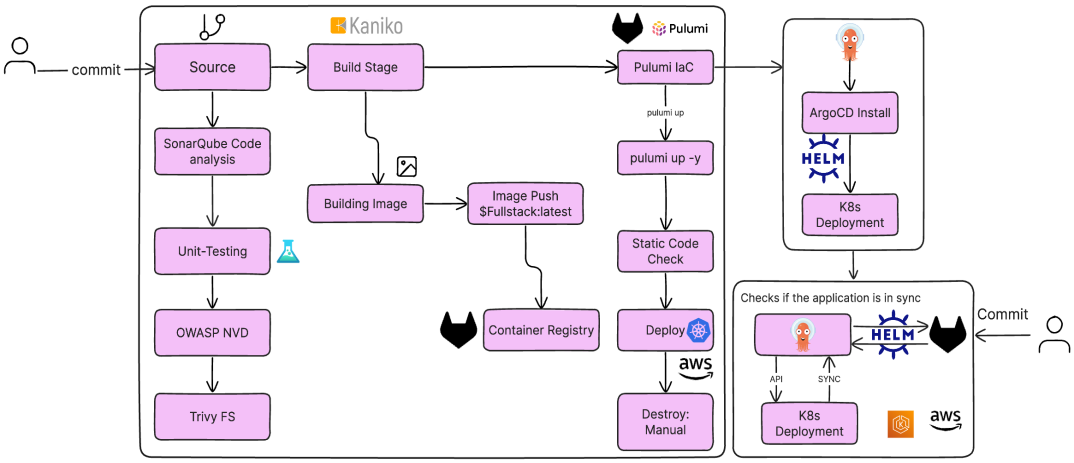

Figure 3.7: Pipeline Workflow using Pulumi

3.7 Kubernetes Cluster details

The Kubernetes deployment using Terraform and Pulumi creates a bunch of resources, roles, and policies for creating a cluster with two worker nodes. The Kubernetes version is 1.29 and is deployed in the AWS EU Ireland region (eu-west-1). Firstly, two different Identity and access management (IAM) roles are defined: master and worker. The master role is required for the

EKS control plane, which allows it to perform essential actions for cluster management. Furthermore, specific policies are attached to the master role and provide permissions needed by the EKS control plane for managing the cluster and services. Additionally, the worker role is required for the worker nodes, allowing them to interact with the worker node instance. The policies attached to the worker node allow the EC2 instances to join the cluster and access the container registry to manage network interfaces. Figure 3.8 illustrates the associated policies and the IAM roles created by IaC.

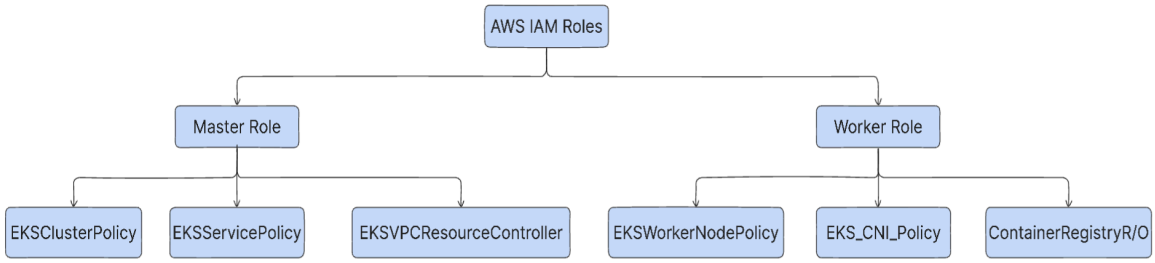

Figure 3.8: Master and Worker Roles in Amazon Elastic Kubernetes Service

Next, an EKS cluster resource is defined with a specific name and VPC configuration, and it specifies the master role Amazon Resource Names (ARN) for the cluster's control plane. Furthermore, a node group is created that groups two worker nodes that are two EC2 instances. Finally, two more resources are created to define access to the EKS cluster and associate an access policy for the user account. As a result, 13 resources are created for a fully functional EKS cluster with two worker nodes. This EKS cluster contains two 3 namespaces, Argo CD, Prometheus, and push-based GitOps; the EKS architecture is illustrated in Figure 3.9. Appendix A.1.1 and A.1.2 are the source codes for Pulumi and Terraform to deploy EKS infrastructure in AWS.

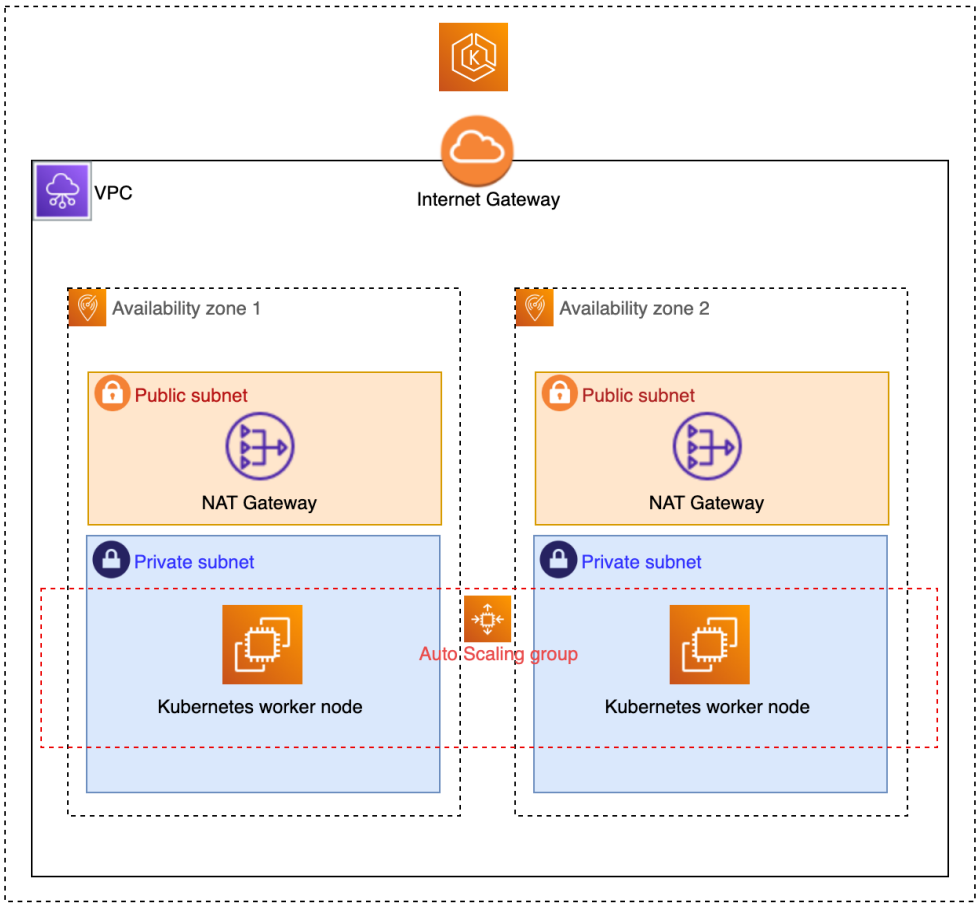

Figure 3.9: EKS Architecture for a Functional K8s Cluster and Two Worker Nodes

3.8 CI Pipeline

Section 3.6 explains the overall pipeline workflow on a high level. This section breaks down the CI and CD pipelines based on the specific tools used to give a better overview of the pipeline workflow.

Figure 3.10 illustrates the CI workflow for the Node.js application and shows various stages and relevant jobs.

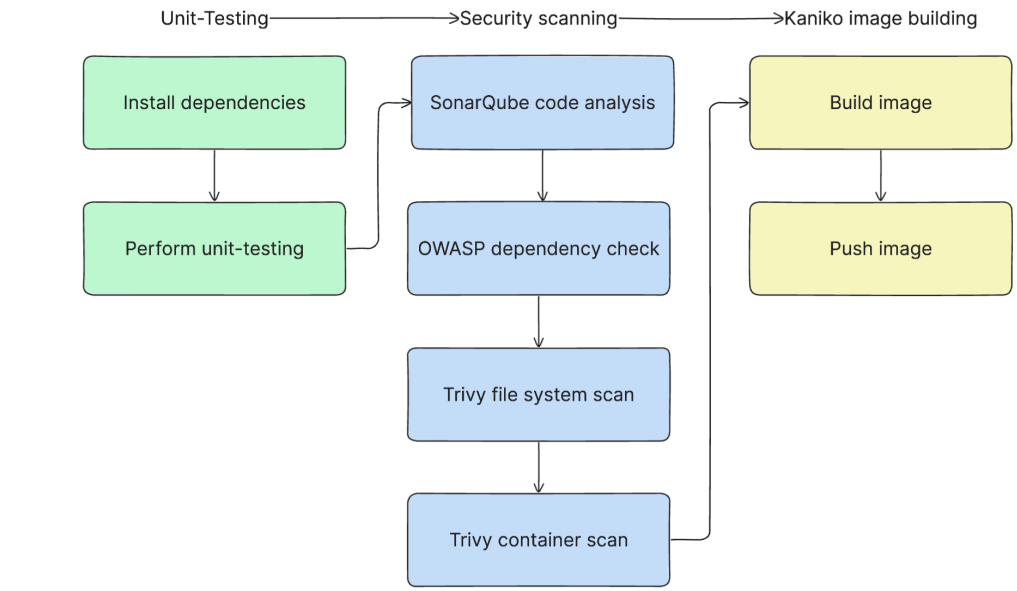

Figure 3.10: CI Pipeline Stages and Jobs

3.9 CD Pipeline

Once the CI pipeline is executed and an image is pushed to the container registry, the CD pipeline and the relevant IaC repositories are invoked.

3.9.1 Terraform

Figure 3.11 illustrates the CD pipeline workflow for the K8s deployment using Terraform. In this case, there are five stages: the terraform plan, apply, installation of Argo CD and Helm chart, and cleanup. The remote state is managed by GitLab native terraform state management, and the AWS provider version is v5.44.0.

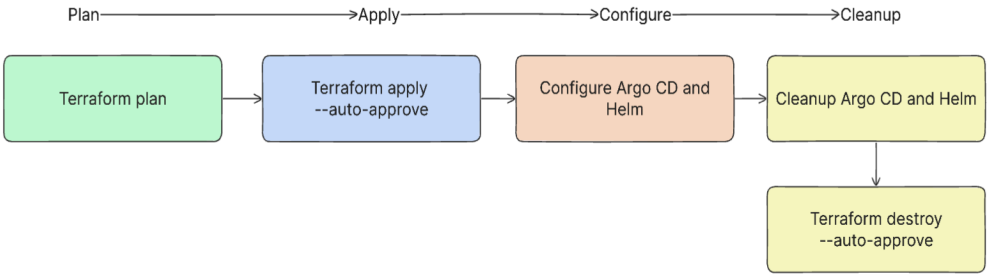

Figure 3.11: Terraform CD Pipeline Stages and Jobs

3.9.2 Pulumi

Figure 3.12 illustrates the CD pipeline workflow, stages, and relevant jobs for the Kubernetes deployment using Pulumi. In this case, there are five stages in the pipeline: the pulumi preview, pulumi up, installation of Argo CD and Helm chart, cleanup of the application, and destruction of the resources. The remote state management is set to AWS S3 bucket since GitLab doesn't provide native support for Pulumi state management. The AWS provider version is v6.32.0.

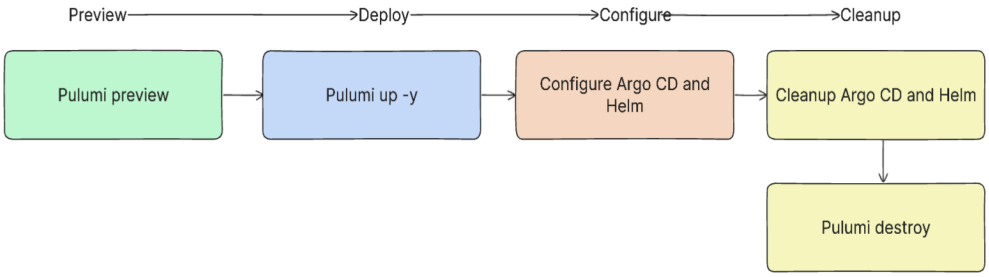

Figure 3.12: Pulumi CD Pipeline Stages and Jobs

3.10 Application Deployment using Helm

Helm chart packages the application into charts, making the deployment of the application in the Kubernetes cluster easier. As described in Section 3.4, a Helm chart is required in a different repository that is in a declarative syntax and defines essential values such as replica count and the image repository. Therefore, the application is packaged into a Helm chart and pushed to the GitLab package registry. As every project repository, including the container image registry, is private in this thesis, an additional layer of security is added. Therefore, the associated GitLab secrets to pull the image have also been defined in the Helm chart.

The values in the Helm chart can be changed to manage and scale the deployed application and Kubernetes infrastructure components, such as ReplicaSet. Appendix A.1.3 shows the Helm chart that is built for application deployment.

3.11 Why are Other Tools not Chosen?

This thesis uses a set of specific tools, technologies, and platforms and is described in Section 3.1. For a number of reasons, other tools were not chosen

for the research.

3.11.1 IaC and Configuration Management Tools

From a preliminary review of the tools and platforms, Ansible, Chef, and Puppet can be excluded from this research because neither of these tools is designed for GitOps workflows compared to IaC tools such as Terraform and Pulumi. Ansible, Chef, and Puppet are utilized primarily for configuration management, as mentioned in Section 2.4.2. They are not used for GitOps approaches because they focus on defining a set of steps to achieve a desired state rather than declaring the state itself. One such example is an Ansible playbook. This makes it challenging to implement GitOps practices since the major GitOps principles state utilizing declarative syntax. As a result, there is no explicit representation of the desired end state in the version-controlled configuration files.

On the other hand, IaC tools like Terraform and Pulumi define IaC in a readable way and follow a declarative approach. This enables developers to verify infrastructure together with application code via version control. Furthermore, Chef and Puppet use an agent-based methodology, which implies that a master agent managing their workflows is necessary for them to operate. The complexity increases as a result. This thesis uses a framework that starts with a continuous integration (CI) pipeline for application code and progresses to a continuous deployment (CD) pipeline that installs the application using Helm chart and launches a fully functional K8s cluster, as illustrated in Figure 3.4. As a result, using an agent-based method would be difficult and complicate the process. On the other hand, Ansible initiates an SSH connection to the hosts on port 22 so that the Ansible script may run [69]. Establishing an SSH connection to the K8s cluster endpoint while deploying a Kubernetes cluster in AWS is not feasible because SSH is typically used for accessing individual nodes in a cluster, not the Kubernetes cluster Application programming interface (API) itself. As a result, Ansible cannot be used to deploy apps, and using Ansible to deploy applications is not recommended for K8s.

Furthermore, Terraform and Pulumi provide native integrations with CI/CD platforms, enabling seamless infrastructure automation. Ansible, Chef, and Puppet require complex setups and custom scripting to achieve similar flexibility for GitOps. As a result, they excel in configuration management scenarios, but their lack of native support for GitOps principles makes them less suitable.

3.11.2 CI/CD Tool

Table 3.2 represents the differences in the significant CI/CD tools. This thesis utilizes the GitLab CI/CD tool due to the benefits mentioned in the table.

Table 3.2: Comparison of CI/CD Tools

Feature	**CircleCI**	**Jenkins**	**GitLab CI/CD**
Hosting	Cloud	Self-hosted	Self-hosted/PaaS
Ease of Use	Easy setup	Complex	User-friendly
Integration	GitHub, Bitbucket	Requires plugins	GitLab ecosystem
Native VCS	×	×	✓
Extensibility	Limited	Highly extensible	Comprehensive
Scalability	Highly scalable	Scalable	Scalable
Docker Support	✓	Plugin required	✓
Kubernetes	✓	Plugin required	✓
Image registry	×	×	✓
Pricing	Freemium	Open source	Freemium
Community	Strong	Strong	Strong

3.11.3 GitOps Operator

Table 3.3 shows the differences in the GitOps operators. This thesis utilizes Argo CD due to the benefits mentioned in the table. Furthermore, Argo CD, according to the 2023 User survey results, received significant positive feedback and production environment adoption rates [70].

Table 3.3: Comparison of Argo CD, Flux v2, and Jenkins X

Parameter	**Argo CD**	**Flux v2**	**Jenkins X**
Auto-Sync	✓	✓	✓
Runs on K8s natively	✓	✓	✓
Multi-tenancy	✓	✓	×
Multi-cluster	✓	✓	✓
User-interface	✓	×	Read-only web UI
Native web UI	✓	×	✓
Helm auto-heal	✓	×	×
CI	×	×	✓
CD	✓	✓	✓
Market Presence	Strong	Growing	Growing

3.11.4 Building Docker Containers

Building docker containers is an essential part of the framework, and in this thesis, Kaniko is utilized instead of Docker in Docker (DinD). It is necessary to understand the benefits of Kaniko and why it is used in the framework, as illustrated in Figure 3.4. Table 3.4 explains the differences between DinD and Kaniko. Kaniko requires no docker daemon and is resource-efficient. Additionally, Kaniko does not require privileged access, making it very secure.

Table 3.4: Comparison of Docker-in-Docker (DinD) and Kaniko

Feature	**DinD**	**Kaniko**
Docker Daemon	✓	×
Builds Docker Images	✓	✓
Docker Socket Required	✓	×
Privileged access	✓	×
Security	Less secure	More secure
Performance	Slower	Faster
Resource Consumption	Higher	Lower
Compatibility	Limited	Broad
Cloud-native	×	✓
Dependency	Docker	None

As a result, due to the above-mentioned differences in the tools, a set of tools has been chosen in the thesis for setting up the environment and performing experiments with respect to the technical environment. The chosen tools are described in Section 3.1. To summarize, this chapter thoroughly explains the related research areas, platforms, tools, and technologies mainly used in DevOps. It aims to build a foundation of important concepts that are essential to comprehend the following chapters, including the design of technical setup and architecture of AWS resources.

This chapter presented a contribution in the form of a comprehensive development of a controlled GitOps environment, utilizing GitLab as the DevSecOps platform and AWS as the public cloud platform. The architecture illustrations of AWS resources represent a best practice for designing them, such as the SonarQube EC2 instance. Additionally, combining open-source security tools such as Trivy, SonarQube, and Kubescape facilitated the formulation of best DevSecOps practices in CI/CD pipelines. Furthermore, implementing an extensive CI/CD framework with Kubernetes, Kaniko, Helm, and Argo CD supports the measurements outlined in the research.

Chapter 4

Results and Analysis

In this chapter, the results and analysis are presented. This chapter extends the results for sub-question 1, sub-question 2 and sub-question 3 as described in Section 1.2.

To comprehend the effect of Infrastructure as Code tools and GitOps methods within the context of CI/CD pipelines using GitLab and AWS, the main research question to answer is:

- How does implementing GitOps and Infrastructure as Code principles impact deployment efficiency, reliability, and security in real-world software development environments?

To summarize, in sub-questions 1 and 2, data are collected for 200 pipelines and 30 pipeline runs, respectively, based on each designed scenario to compare Infrastructure as Code tools and GitOps methods within the context of automated pipelines. Therefore, as described in the Central Limit Theorem (CLT) in Section 1.5, these samples can be assumed to be normally distributed, and the parameter to be estimated is the difference between the means of the two samples.

Hence to quantify the uncertainty in the parameter, the Confidence Interval (CI) can be calculated as:

$$C.I = (\bar{X}_1 - \bar{X}_2) \pm ZS_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

Where $\bar{X}_1$ and $\bar{X}_2$ denote the mean times of the Pulumi and Terraform samples, or Push and Pull-based samples respectively and the Z-Value is ≈ 1.96 for 95% confidence, and, n_1 and n_2 are the sample sizes of Pulumi and Terraform or

Push and Pull-based GitOps methods respectively.

The pooled estimate S_p of the common standard deviation is calculated as:

$$S_p = \sqrt{\frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}}$$

Where S_1 and S_2 are the sample standard deviation of the Pulumi and Terraform sample or Push and Pull-based groups, respectively.

All the above equations are valid only when the ratio of the sample variances, $\frac{s_1^2}{s_2^2}$, is in between 0.5 and 2.

The research question is split into three sub-questions, and they are solved and analyzed in sequence:

4.1 Sub-Question 1

- How does adopting Infrastructure as Code tools affect the efficiency and reliability of infrastructure deployment processes?

This sub-question aims to measure the speed of AWS EKS deployments within the context of CI/CD pipelines using IaC tools, Terraform and Pulumi. It assists businesses in choosing the best IaC tool in terms of infrastructure deployment speeds when combined with GitLab CI/CD pipelines. The sub-question is solved using three scenarios and is designed to replicate the business point of view.

4.1.1 Scenario 1: Comparison of IaC Deployment Stages

As described in Section 3.9, Pulumi and Terraform pipelines both have a set of stages in their respective pipelines. This scenario aims to compare the speeds of the deployment stages, which are “deploy” and “apply” in Pulumi and Terraform, respectively.

- The time taken for the “Infrastructure Deployment” stages has been compared 200 times for each IaC tool. Each set of 50 pipelines deploys an AWS EKS cluster and has an increase in the number of worker nodes by 2. Therefore, the number of worker nodes is 2,4,6 and 8.

1. EKS cluster with 2 worker nodes: The average time for Pulumi's deploy and Terraform's apply stages are **10 minute 20 seconds** and **10 minute 11 seconds**, respectively.

$$
\begin{aligned}
n_1 &= n_2 = 50 \\
S_1 &= 1minute, 3seconds = 63seconds \\
S_2 &= 49seconds
\end{aligned}
$$

As $\frac{{S1}^2}{{S2}^2} \in (0.5, 2)$, the below equations are valid.

The pooled estimate of the common standard deviation is calculated as follows:

$$S_p = \sqrt{\frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}}$$

Substituting the values:

$$S_p = \sqrt{\frac{(49) \times 3969 + (49) \times 2401}{98}} = \sqrt{3185} \approx 56.43seconds$$

$$
\begin{aligned}
\bar{X}_1 &= 10minutes, 20seconds = 620seconds \\
\bar{X}_2 &= 10minutes, 11seconds = 611seconds \\
S_p &= 56.43seconds \\
n_1 &= n_2 = 50 \\
Z &= 1.96
\end{aligned}
$$

The confidence interval for the difference between the means is calculated as:

$$C.I = (\bar{X}_1 - \bar{X}_2) \pm ZS_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

Substituting the values:

$$(620 - 611) \pm 1.96 \times 56.43 \times \sqrt{\frac{1}{50} + \frac{1}{50}}$$

$$(620 - 611) \pm 1.96 \times 56.43 \times 0.2$$

$$C.I = 9 \pm 22.12$$

Therefore, the Confidence Interval (CI) in the case of 2 worker nodes is (-13.12, 31.12). With 95% confidence, the difference in the mean times of Pulumi and Terraform is between -13.12 and 31.12 seconds. Since the Confidence Intervals contain zero or the null value, it can be concluded that there is no statistically meaningful or statistically significant difference between Pulumi and Terraform.
This means that the mean time of Pulumi could be greater than, less than, or equal to the meantime of Terraform, and there is not enough information to determine which one is the case.

2. EKS cluster with 4,6 and 8 worker nodes: As $\frac{S1^2}{S2^2} \in (0.5, 2)$, the equation for the pooled estimate of the common standard deviation and the confidence interval is valid. Table 4.1 shows the variables based on which the calculations are performed. Table 4.2 shows the calculated values of the confidence interval of the difference in mean times for 4, 6 and 8 worker nodes for Pulumi and Terraform.

Table 4.1: Descriptive Statistics on Variables measured

	4	**6**	**8**
$\frac{S1^2}{S2^2}$	0.76	1.05	0.84
(n_1, n_2)	(50, 50)	(50, 50)	(50, 50)
$(\bar{x}_1, \bar{x}_2)$ in sec	(616, 613)	(616, 597)	(628, 607)
(S_1, S_2) in sec	(48, 63)	(54, 51)	(50, 59)
S_P in sec	56	52.52	54.68

Table 4.2: Confidence Intervals in case of 4,6 and 8 Worker Nodes

	4	**6**	**8**
95% C.I	(−18.95, 24.95)	(-1.58, 39.58)	(-0.43, 42.43)

Since the Confidence Intervals in cases of 4,6,8 worker nodes contain zero or the null value, it can be concluded that there is no statistically meaningful or statistically significant difference between Pulumi and Terraform. This means that the mean time of Pulumi could be greater than, less than, or

equal to the meantime of Terraform, and there is not enough information to determine which one is the case. As a result, in the first scenario, it cannot be concluded which tool, Terraform or Pulumi, performs better when the speeds of deployment stages are compared.

4.1.2 Scenario 2: Comparison of CI/CD Framework for each IaC Tool

This scenario compares the average pipeline run-time for the entire framework as described in Section 3.4, using Pulumi and Terraform, respectively. As a result, this scenario aims to assist businesses in picking the best-suited IaC tool among Pulumi and Terraform in cases where continuous application integration and infrastructure deployment are required within the context of CI/CD pipelines.

- The average time taken for the CI/CD framework as described in Section 3.4 is compared 200 times for each IaC tool. Each set of 50 pipelines runs a continuous integration pipeline and deploys an AWS EKS cluster, with an increase in the number of worker nodes by 2. Therefore, the number of worker nodes is 2,4,6 and 8. The CI pipeline consists of a downstream pipeline that triggers the pipelines of IaC repositories. The cleanup stages of the IaC tools are excluded in this scenario.

CI/CD Framework - AWS EKS with 2,4,6 and 8 worker nodes

The confidence interval has been calculated based on the equations described in Appendix B. As $\frac{S1^2}{S2^2} \in (0.5, 2)$, the equation for the pooled estimate of the common standard deviation and the confidence interval is valid. Table 4.3 shows the variables based on which the calculations are performed. Table 4.4 shows the calculated values of the confidence interval of the difference in mean times for 2, 4, 6 and 8 worker nodes for Pulumi and Terraform.

Table 4.3: Descriptive Statistics on Variables measured

	2	4	6	8
$\frac{S_1^2}{S_2^2}$	0.75	1.39	1.02	1.21
(n_1, n_2)	(50, 50)	(50, 50)	(50, 50)	(50, 50)
$(\bar{x}_1, \bar{x}_2)$ in sec	(1064, 1037)	(1053, 1010)	(1022, 1000)	(1038, 1006)
(S_1, S_2) in sec	(46, 53)	(74, 53)	(49, 48)	(45, 37)
S_P in sec	49.62	64.36	48.48	41.23

Table 4.4: Confidence Intervals in case of 2, 4, 6, and 8 Worker Nodes

	2	4	6	8
95% C.I	(7.6, 46.4)	(17.77, 68.23)	(3, 41)	(15.84, 48.16)

With 95% confidence, the difference in the mean times of Pulumi and Terraform is between 7.6 to 46.4 for 2 worker nodes, 17.77 to 68.23 seconds for 4 worker nodes, 3 to 41 seconds for 6 worker nodes, and 15.84 to 48.16 for 8 worker nodes.

This means the difference in mean times between Pulumi and Terraform is always positive. As a result, the time taken for Pulumi to deploy EKS in AWS based on the established CI/CD framework is always greater than Terraform. Therefore, in the second scenario, it can be concluded that when Pulumi and Terraform are compared with respect to the established CI/CD framework, Terraform performs better and has a lower pipeline run time.

4.1.3 Scenario 3: Comparison of CI/CD Framework for each IaC Tool with Infrastructure existing in AWS

This scenario compares the average pipeline run-time for the entire framework as described in Section 3.4, using Pulumi and Terraform, respectively, while the AWS EKS infrastructure exists. As a result, this scenario aims to assist businesses in picking the best-suited IaC tool among Pulumi and Terraform in cases where pipelines are run to continuously deploy and manage applications into the existing infrastructure or in cases where pipelines are run to match the desired infrastructure state with the current infrastructure state.

- The average time taken for the CI/CD framework as described in Section 3.4 is compared 200 times for each IaC tool when the infrastructure is already present in AWS. The cleanup stages of the IaC tools are excluded in this scenario. In this scenario, the increase in the number of worker nodes does not matter since the infrastructure must be present in AWS beforehand.

CI/CD Framework when the EKS infrastructure exists in AWS

The confidence interval has been calculated based on the equations described in Appendix B. As $\frac{S1^2}{S2^2} \in (0.5, 2)$, the equation for the pooled estimate of the common standard deviation and the confidence interval is valid. Table

4.5 shows the variables based on which the calculations are performed. Table 4.6 shows the calculated values of the confidence interval for the difference in mean times for Pulumi and Terraform.

Table 4.5: Descriptive Statistics on Variables measured

	CI/CD Framework - Existing AWS Infrastructure
$\frac{S_1^2}{S_2^2}$	1.12
(n_1, n_2)	(200, 200)
$(\bar{x}_1, \bar{x}_2)$ in sec	(504, 454)
(S_1, S_2) in sec	(35, 33)
S_P in sec	34.01

Table 4.6: Calculated Confidence Intervals for Differences in Mean Times for Pulumi and Terraform in Case of Existing Infrastructure on AWS

	CI/CD Framework - Existing AWS Infrastructure
95% C.I	(43.34, 56.6)

Therefore, the Confidence Interval (CI) is (43.34, 56.6). With 95% confidence, the difference in the mean times of Pulumi and Terraform is between 43.34 and 56.6 seconds.

This means the difference in mean times between Pulumi and Terraform is always positive. As a result, the time taken for Pulumi to match the desired infrastructure state with the current infrastructure state based on the established CI/CD framework is always greater than Terraform. As a result, in the third scenario, it can be concluded that when Pulumi and Terraform are compared with respect to the established CI/CD framework and existing AWS infrastructure, Terraform performs better and has a lower pipeline run-time.

4.2 Sub-Question 2

- To what extent are GitOps methods, such as push and pull methods, efficient in software and infrastructure modifications?

This sub-question aims to measure the speed of application upgrades, downgrades and infrastructure management within the context of CI/CD pipelines using pull and push-based GitOps methods. The pull and push-based GitOps methods are explained in Section 2.6.4. It assists businesses

in choosing the best GitOps method when there is a need for continuous application upgrades, downgrades, and infrastructure scaling combined with GitLab CI/CD pipelines. The GitOps operator in the pull-based method utilized in this thesis is Argo CD. The sub-question is solved using three scenarios and is designed to replicate the business point of view.

4.2.1 Scenario 1: Comparison of GitOps Methods during Rolling Updates

New features in the application are committed using Git. This is also known as rolling updates. In this scenario, the CI pipeline, as described in Section 3.8, is run, and a downstream pipeline is triggered that invokes the Helm chart pipeline and applies the changes in the push-based method. Whereas the GitOps operator automatically picks the changes as soon as the Helm chart repository is invoked. This scenario assists businesses in choosing the best GitOps method when there is a requirement for application upgrades within the context of GitLab CI/CD pipelines.

- The time taken for the application upgrades, also called rolling updates, is compared 30 times for each GitOps method. For both pull-and-push-based GitOps methods, the precise timestamp of "git commit" and the timestamp for application upgrades are compared.

The confidence interval has been calculated based on the equations described in Appendix B. As $\frac{S1^2}{S2^2} \in (0.5, 2)$, the equation for the pooled estimate of the common standard deviation and the confidence interval is valid. Table 4.7 shows the variables based on which the calculations are performed. Table 4.8 shows the calculated values of the confidence interval for differences in mean times for Push and Pull based GitOps methods during application rolling updates.

Table 4.7: Descriptive Statistics on Variables measured

	Comparison of GitOps Methods during Rolling Updates
$\frac{S_1^2}{S_2^2}$	1
(n_1, n_2)	(30, 30)
$(\bar{x}_1, \bar{x}_2)$ in sec	(173, 148)
(S_1, S_2) in sec	(25, 25)
S_P in sec	25

Table 4.8: Calculated Confidence Intervals for Differences in Mean Times for Push and Pull-Based GitOps Methods in Case of Rolling Updates

	GitOps Methods in Case of Rolling Updates
95% C.I	(12.35, 37.65)

Therefore, the Confidence Interval (CI) is (12.35, 37.65). With 95% confidence, the difference in the mean times of Push and Pull-based GitOps methods is between 12.35 and 37.65 seconds.

This means the difference in mean times between Push and pull-based GitOps methods is always positive. As a result, the time taken for the push-based GitOps method to perform application rolling updates is always greater than the pull-based GitOps method. As a result, in the first scenario, it can be concluded that when Push and Pull-based GitOps methods are compared for application upgrades (Rolling Updates), the Pull-based GitOps method performs better and upgrades the application faster.

4.2.2 Scenario 2: Comparison of GitOps Methods during Application Rollback

The application is rolled back to a previous version using Git. The command "git revert <commit-hash>" is utilized to roll back the changes. In this scenario, the CI pipeline, as described in Section 3.8, is run, and a downstream pipeline is triggered that invokes the Helm chart pipeline and applies the changes in the push-based method. Whereas the GitOps operator automatically picks the changes as soon as the Helm chart repository is invoked. This scenario assists businesses in choosing the best GitOps method when there is a requirement for application rollbacks within the context of GitLab CI/CD pipelines.

- The time taken for the application rollbacks is compared 30 times for each GitOps method. For both pull-and-push-based GitOps methods, the precise timestamp of "git commit" and the timestamp for application rollbacks are compared.

The confidence interval has been calculated based on the equations described in Appendix B. As $\frac{S1^2}{S2^2} \in (0.5, 2)$, the equation for the pooled estimate of the common standard deviation and the confidence interval is valid. Table 4.9 shows the variables based on which the calculations are performed. Table 4.10

shows the calculated values of the confidence interval for differences in mean times for Push and Pull based GitOps methods during application rollbacks.

Table 4.9: Descriptive Statistics on Variables measured

	Comparison of GitOps Methods during Application Rollbacks
$\frac{S_1^2}{S_2^2}$	1.2
(n_1, n_2)	(30, 30)
$(\bar{x}_1, \bar{x}_2)$ in sec	(168, 144)
(S_1, S_2) in sec	(18, 16)
S_P in sec	17.03

Table 4.10: Calculated Confidence Intervals for Differences in Mean Times for Push and Pull-Based GitOps Methods in Case of Application Rollbacks

	GitOps Methods in Case of Application Rollbacks
95% C.I	(15.39, 32.61)

Therefore, the Confidence Interval (CI) is (15.39, 32.61). With 95% confidence, the difference in the mean times of Push and Pull-based GitOps methods is between 15.39 and 32.61 seconds.

This means the difference in mean times between Push and pull-based GitOps methods is always positive. As a result, the time taken for the push-based GitOps method to perform application rollbacks is always greater than the pull-based GitOps method. As a result, in the second scenario, it can be concluded that when Push and Pull-based GitOps methods are compared in the case of application rollbacks, the Pull-based GitOps method performs better and rolls back the application at a faster rate.

4.2.3 Scenario 3: Comparison of GitOps Methods during K8s Pod Scaling

The declarative syntax of the Helm chart is modified directly to scale resources such as pods in Kubernetes. When the Helm chart is modified, the GitOps operator instantly picks up the changes, whereas, in the Push-based GitOps method, a pipeline is run to manage the K8s pods. The measurement is based on how fast the infrastructure scales for both pull- and push-based GitOps methods.

- The time taken for the application rollbacks is compared 30 times for each GitOps method. For both pull-and-push-based GitOps methods, the precise timestamp of "git commit" and the timestamp for application rollbacks are compared.

The confidence interval has been calculated based on the equations described in Appendix B. As $\frac{S1^2}{S2^2} \in (0.5, 2)$, the equation for the pooled estimate of the common standard deviation and the confidence interval is valid. Table 4.11 shows the variables based on which the calculations are performed. Table 4.12 shows the calculated values of the confidence interval for differences in mean times for Push and Pull based GitOps methods during pod scaling.

Table 4.11: Descriptive Statistics on Variables measured

	Comparison of GitOps Methods during Pod Scaling
$\frac{S_1^2}{S_2^2}$	1.65
(n_1, n_2)	(30, 30)
$(\bar{x}_1, \bar{x}_2)$ in sec	(29, 3)
(S_1, S_2) in sec	(9, 7)
S_P in sec	8.06

Table 4.12: Calculated Confidence Intervals for Differences in Mean Times for Push and Pull-Based GitOps Methods in Case of Pod Scaling

	GitOps Methods during Pod Scaling
95% C.I	(21.92, 30.08)

Therefore, the Confidence Interval (CI) is (21.92, 30.08). With 95% confidence, the difference in the mean times of Push and Pull-based GitOps methods is between 22.76 and 29.24 seconds.

This means the difference in mean times between Push and pull-based GitOps methods is always positive. As a result, the time taken for the push-based GitOps method to perform pod scaling is always greater than the pull-based GitOps method. As a result, in the third scenario, it can be concluded that when Push and Pull-based GitOps methods are compared in the case of K8s pod scaling, the Pull-based GitOps method performs better and scales the K8s pods at a faster rate.

4.3 Sub-Question 3

This sub-question aims to suggest the relevant open-source security tools that can be integrated into the automated CI/CD pipelines for optimal security. The research in this sub-question explores the importance of addressing security vulnerabilities early in the CI/CD pipelines to prevent potential risks and vulnerabilities from reaching production systems. Businesses may utilize the tools described in Table 4.13 to incorporate robust security measures in the existing DevSecOps pipelines. These tools can be utilized in any kind of DevOps or CI/CD platform and, therefore, don't have a dependency on a single platform.

- SQ3: What are the best security practices in GitOps, and which tools can be employed to integrate security in the CI/CD pipelines?

Integrating security practices within DevOps pipelines, also known as DevSecOps, is essential for ensuring the integrity of software development processes. By utilizing robust security tools such as OWASP (Open Web Application Security Project), SonarQube, Trivy and Kubescape, alongside incorporating data sources like the National Vulnerability Database (NVD), which collectively catalogue over 250,000 vulnerabilities, development teams can proactively identify and address security flaws throughout the development lifecycle.

Tools such as SonarQube can be self-hosted for better management and security. When vulnerabilities are detected, the pipelines can be stopped from running and the application image deployed. These tools offer comprehensive security checks throughout the CI/CD pipeline stages, facilitating automated code reviews, static code analysis, container image scanning, and Kubernetes security assessments.

As a result, by integrating such security measures in the CI/CD pipelines, businesses can build and deploy secure applications and infrastructure while effectively mitigating security risks. Table 4.13 summarizes the set of security tools which can be incorporated into CI/CD pipelines and Kubernetes clusters.

Table 4.13: Security Tools and their Functionalities

S No.	Tool	Description
1	OWASP	Web app security
2	SonarQube	Code quality
3	Trivy	Container security
4	Kubescape	Kubernetes security
5	NVD	Vulnerability database

4.4 Discussion based on Metrics

The results are highly reliable since the experiments are performed in isolated environments for each scenario within the sub-questions 1 and 2. In the experiments in sub-questions 1 and 2, in which Pulumi, Terraform, and Pull-based, Push-based GitOps methods are compared, the GitLab runners that are essential to run the pipelines are two isolated EC2 instances in AWS. As a result, except for the influence on the EC2 instances from the AWS cloud platform, there is no external influence on network latency or compute resource utilization on these runners. The experiments are also conducted with the GitLab CI/CD platform as the foundation, which is self-hosted and has High Availability (HA) architecture. Therefore, there is no influence of platform slowness or latency issues compared to publicly available "https://gitlab.com/".

4.4.1 Sub-question 1

To summarize the first scenario of sub-question 1, when a single stage called "deploy" and "apply" in Pulumi and Terraform are compared to each other 200 times, with an increase in the number of worker nodes by 2, the confidence intervals with 95% confidence, contained a null value or a zero, which means that there is no statistically meaningful or a statistically significant difference between Pulumi and Terraform. Therefore, when combined with automated CI/CD pipelines in GitLab, businesses may rely on either of these tools when it comes to picking one in terms of deployment speeds.

To summarize the second scenario of sub-question 1, when the overall CI/CD framework is compared 200 times for both Pulumi and Terraform, with an increase in the number of worker nodes by 2, with 95% confidence, Pulumi had a higher pipeline runtime in all cases. This means that Terraform is a better-performing tool in terms of speed when it is combined with the

continuous integration pipeline. As a result, when combined with automated CI/CD pipelines in GitLab, businesses may rely on Terraform in a use case where continuous integration pipelines for application testing, building, and development are run along with the deployment of the application into a cloud infrastructure resource.

To summarize the third scenario of sub-question 1, when the overall CI/CD framework is compared 200 times with existing infrastructure on AWS, with 95% confidence, Pulumi had a higher pipeline runtime in all cases. This means that Terraform is a better-performing tool in terms of speed when it is combined with the continuous integration pipeline and existing AWS infrastructure. As a result, when combined with automated CI/CD pipelines in GitLab, and with existing infrastructure on AWS, in comparison to Pulumi, businesses may rely on Terraform since it is faster at comparing the desired state of the infrastructure with the current state and likewise, it can deploy the applications at a faster speed in existing infrastructure.

Both tools are 100% reliable. This means that in all three scenarios, Pulumi and Terraform have deployed the AWS EKS infrastructure successfully without any failure in any of the stages which are described in Section 3.9.

4.4.2 Sub-question 2

To summarize the first scenario of sub-question 2, when the speeds of application upgrades (Rolling Updates) were measured 30 times, with 95% confidence, the Push-based GitOps method had higher application upgrade runtime. This means that Argo CD, when utilized for the Pull-based method and as a GitOps operator, performs application upgrades faster than those of the traditional Push-based GitOps method. As a result, businesses may rely on the Pull-based GitOps method to increase the speeds of application upgrades when combined with CI/CD pipelines.

To summarize the second scenario of sub-question 2, when the speeds of application rollbacks were measured 30 times, with 95% confidence, the Push-based GitOps method had a higher application rollback time. This means that Argo CD, when utilized for the Pull-based method and as a GitOps operator, performs application rollbacks faster than those of the traditional Push-based GitOps method. Therefore, businesses may rely on the Pull-based GitOps method to increase the speeds of application rollbacks when combined with CI/CD pipelines.

To summarize the third scenario of sub-question 2, when the speeds of scaling of the EKS pods were measured 30 times, the Push-based GitOps

method had higher K8s pod scaling time. This means that Argo CD, when utilized for scaling the pods of the K8s cluster and as a GitOps operator, scales the pods faster than those of the traditional Push-based GitOps method. In this case, as described in Section 2.6.2, the timing reconciliation of Argo CD is set to 1 second, which means that Argo CD picks up the changes within the Git repository just after a second of "git push". Therefore, businesses may rely on the Pull-based GitOps method to increase the speeds of infrastructure scaling and modifications when combined with CI/CD pipelines.

Chapter 5

Discussion

Understanding the best-suited Infrastructure as a Code tool, GitOps operators, and methods for CI/CD and production-grade environments is a crucial task. It is not just about technical capabilities but also about aligning with the business's specific needs. This section will delve into the solutions that compare the setup of the IaC tools, GitOps operators, integration with CI/CD environments, upgrade processes, deployment efforts, costs, and performance to help companies and businesses make an informed decision that best serves their needs.

When it comes to choosing an Infrastructure-as-Code tool within the context of CI/CD environments, Pulumi and Terraform are the available tools that can deploy all kinds of Infrastructure in the major public cloud platforms, such as AWS, Microsoft Azure, and Google Cloud Platform (GCP). Currently, the Infrastructure-as-Code tool Pulumi is open-source with the Apache 2.0 license, and the Software Development Kit (SDK) is open for the public to view and download. Additionally, it is easier for businesses to include Pulumi in their infrastructure management practices because Pulumi enables developers to utilize any programming language to write the code.

As a result, a developer aware of a programming language may quickly deploy Infrastructure using a language they know. As a result, Pulumi allows multiple teams, including developers and DevOps engineers, to create code for the infrastructure using a language they are familiar with. However, Terraform requires the code to be in a Domain-Specific Language (DSL). Furthermore, Terraform is no longer open-source and has a Business Source License (BSL). Terraform may not be the best pick for practitioners and businesses that rely only on open-source Infrastructure-as-Code tools.

Both tools are free to use and integrate seamlessly with multiple CI/CD

platforms. Pulumi also provides testing functionality such as unit, property, and integration, while Terraform primarily supports integration testing. Additionally, Pulumi offers seamless secret management, encrypting sensitive values at rest and in transit. As a result, as of May 2024, Pulumi provides more features than Terraform.

Terraform and Pulumi are easy to set up locally and on CI/CD platforms such as GitLab. With remote backends in place, major CI/CD platforms such as GitLab provide native support for the Terraform backend, whereas Pulumi lacks native support for the remote backend in GitLab.

To summarize, Terraform may be the right tool for businesses where infrastructure deployment speeds are needed when CI/CD pipelines and platforms are the core of application development and infrastructure deployment. Especially with the integration of GitLab, a prominent DevSecOps platform, and native support for remote backends, Terraform is a proper fit. Whereas, Pulumi may be the right tool for businesses opting for the latest features in Infrastructure as Code, such as native support for testing, secret management, multi-programming language support, and most importantly, an open-source license.

Secondly, utilizing the GitOps concepts, such as version controlling the Infrastructure as Code, is considered good practice in managing the Infrastructure as Code regardless of the tool used. With the help of a Version Control System (VCS), the traceability of the code is seamless. It assists the developers in reusing a previous version of the code without any hassle. However, the core concept of GitOps, which uses a GitOps operator to manage Kubernetes applications, can be an excellent fit for businesses where Kubernetes and container orchestration, when combined with CI/CD pipelines and platforms, are critical technologies for their operations. Open-source GitOps operators such as Argo CD are a great fit when combined with CI/CD pipelines and platforms such as GitLab. GitOps operators facilitate seamless management of Kubernetes applications and their resources, such as ReplicaSets.

GitOps operators such as Argo CD are easy to set up and are open-source tools. With the use of GitOps operators, there is no requirement for “kubectl” and access to the cluster, which means developers may only be required to work on the project repositories rather than performing administrative tasks on the Kubernetes cluster with a privileged level of access.

To summarize, GitOps methods such as the Pull-based deployment method and tools such as Argo CD, combined with CI/CD pipelines and platforms such as GitLab, significantly improve speeds during application upgrades,

rollbacks, and Kubernetes pod scaling. Additionally, these GitOps operators are also easy to set up within the Kubernetes clusters, allowing developers to utilize the concept of GitOps for all types of applications deployed within Kubernetes.

Finally, the best open-source security tools and best practices for CI/CD pipelines are discussed. These significantly add multiple layers of security within the automated CI/CD pipelines, such as container image and file system scanning, static code analysis, vulnerability, and dependency scanning. These assist businesses in detecting vulnerabilities in their code and related dependencies at a very early stage of application development.

Chapter 6

Conclusions and Future work

This chapter outlines the main conclusions, chapter 1-based reflections, and the thesis work's limitations. The future area of research in the field of GitOps and Infrastructure as Code (IaC) in the context of CI/CD pipelines is conveyed in this chapter, which is essential.

6.1 Conclusions

The thesis project delves deeply into the subject areas of GitOps, DevSecOps and Infrastructure as Code (IaC) in CI/CD environments. First, the concepts of CI/CD pipelines, DevSecOps, GitLab, GitOps, Kubernetes, AWS and Infrastructure as Code are explored comprehensively to understand the combination of relevant tools, practices, technologies and platforms. Sub-questions and their related scenarios are defined to mimic a company's perspective of application development and infrastructure deployment, and a comprehensive framework involving a CI/CD pipeline with prominent tools and technologies is built. As a result, within the context of CI/CD pipelines on GitLab, Pulumi and Terraform, two prominent Infrastructure as code (IaC) tools are compared within designed scenarios. Furthermore, GitOps methods, with and without GitOps operator, also known as pull and push-based GitOps methods, are compared to understand the effect of these methods in Kubernetes concerning application upgrades, rollbacks and pod scaling when combined with GitLab CI/CD pipelines. Finally, effective practices for CI/CD pipelines and the top open-source security tools are covered, which aim to add multiple security layers to the automated GitLab CI/CD pipelines.

The thesis evolved to address the sub-questions, and the conclusions of the experiments make it easier to select the ideal GitOps method for

managing Kubernetes infrastructure resources, application management and Infrastructure as a Code tool for automated CI/CD pipelines in GitLab.

6.2 Limitations

This thesis project is solely focused on a single cloud platform, AWS. The results may vary when similar research is performed on other cloud platforms, such as Microsoft Azure and Google Cloud Platform. Additionally, the GitLab platform utilized in this thesis project is a self-hosted platform, and the results may vary when similar research is performed on "www.gitlab.com." Resource utilization, network latency, and auto-scaling of the self-hosted GitLab instance may affect the CI/CD pipeline performance. Furthermore, this thesis does not emphasize the production-grade application in the CI/CD framework.

When conducting the experiments, the Terraform and Pulumi providers chosen were v5.44.0 and v6.32.0, and results may vary based on the latest provider releases. Finally, the GitLab runner, which is essential to run the pipelines, is an EC2 instance with the instance type "t3.xlarge." It offers four vCPUs (Virtual Central Processing Units), 16 GB (Gigabyte) of memory, and 5 Gigabit network performance. The result may vary if the CI/CD pipelines utilize a different instance type. Finally, the thesis utilizes the latest open-source security tools as of May 2024. These tools may evolve with time, and a new set of tools may offer better security features for CI/CD pipelines.

6.3 Future Work

Due to a lack of time, the GitLab runner's observability, monitoring, and system performance were not explored in depth during the experiments. Different IaC tools may show different resource and network utilization. The reliability of the Infrastructure as Code tools can be stressed by simulating networking failures with the help of tools like "tc" (Linux Traffic Control) to add artificial network latency and simulate loss of network packets between the hosted CI/CD runners and AWS services. Reliability can also be stressed by intentionally including invalid configurations and errors, such as incorrect resource definitions in the syntax of Infrastructure as Code. Additionally, the scope of the research may be extended to other public cloud platforms. Furthermore, the cost factors for setting up the framework, GitLab runners, and its associated tools and technologies were not explored in depth. Also, surveys

and interviews can be done with the team of developers within a company to understand their requirements for application development and infrastructure deployment within the context of CI/CD environments.

Furthermore, similar research can be performed with a production-ready application and its continuous integration (CI) pipeline to comprehend the results accurately. Finally, in this thesis, GitLab is the chosen DevSecOps platform, and Argo CD is the GitOps operator. Similar research with other platforms and tools, such as Jenkins and Flux CD, opens a broad research area within the field of GitOps and can be explored.

6.4 Reflections

This thesis project is essential For Scania CV AB, where the research is conducted. The analysis and outcomes assist the company in choosing the most appropriate infrastructure for use as a code tool, GitOps method, security tools, and GitOps operator when combined with GitLab pipelines for Continuous Integration and Continuous Deployment/Delivery. When thorough application development and infrastructure deployment scenarios are considered, the sub-questions in this thesis are meant to mimic the business point of view. When writing the thesis, there's an absence of a thorough comparative study about Infrastructure as Code tools, particularly in the case of GitOps, a new technology that is an extension of DevOps. This study provides a strong basis for understanding the differences between Pulumi and Terraform, GitOps methods, GitOps operators and security tools in a controlled GitOps and CI/CD setting, which will benefit researchers and DevOps practitioners equally.

References

[1] A. Mihelič, S. Vrhovec, and T. Hovelja, "Agile development of secure software for small and medium-sized enterprises," *Sustainability*, vol. 15, no. 1, p. 801, Jan 2023. doi: 10.3390/su15010801 [Page 2.]

[2] T. Hall, "Devops culture." [Online]. Available: https://www.atlassian.com/devops/what-is-devops/devops-culture [Page 2.]

[3] M. S. Khan, A. W. Khan, F. Khan, M. A. Khan, and T. K. Whangbo, "Critical challenges to adopt devops culture in software organizations: A systematic review," *IEEE Access*, vol. 10, p. 14339–14349, 2022. doi: 10.1109/access.2022.3145970 [Page 3.]

[4] 3. introduction to terraform - practical GitOps: Infrastructure management using terraform, AWS, and GitHub actions [book]. ISBN: 9781484286739. [Online]. Available: https://www.oreilly.com/library/view/practical-gitops-infrastructure/9781484286739/html/524191_1_En_3_Chapter.xhtml [Page 3.]

[5] Ramadoni, E. Utami, and H. A. Fatta, "Analysis on the use of declarative and pull-based deployment models on gitops using argo cd," in *2021 4th International Conference on Information and Communications Technology (ICOIACT)*, 2021. doi: 10.1109/ICOIACT53268.2021.9563984 pp. 186–191. [Pages 4, 39, 43, and 44.]

[6] "Aws - sustainability." [Online]. Available: https://sustainability.aboutamazon.com/products-services/the-cloud?energyType=true [Page 5.]

[7] O. College, B. Illowsky, and S. Dean, *Introductory Statistics: Page 400, Chapter 7*, 1st ed. Openstax College, Rice University, 2013. ISBN 1938168208,9781938168208,1947172050,9781947172050,1938168291,9781938168291. [Online]. Available: https://assets.openstax.org/oscms-prodcms/media/documents/IntroductoryStatistics-OP_i6tAI7e.pdf [Pages 7 and 98.]

[8] D. S. Shafer and Z. Zhang, *Beginning Statistics: Page 445, Chapter 9*. Open Textbook Library, 2012, retrieved from https://2012books.lardbucket.org/pdfs/beginning-statistics.pdf. [Page 7.]

[9] ——, *Beginning Statistics: Page 466, Chapter 9*. Open Textbook Library, 2012, retrieved from https://2012books.lardbucket.org/pdfs/beginning-statistics.pdf. [Pages 7, 98, and 99.]

[10] ——, *Beginning Statistics: Page 490, Chapter 9*. Open Textbook Library, 2012, retrieved from https://2012books.lardbucket.org/pdfs/beginning-statistics.pdf. [Pages 7 and 99.]

[11] D. Wadhwa, "What is the relationship between devops and cloud," https://www.ccsinet.com/blog/relationship-devops-cloud/, April 29 2019, cCSI, A Sourcepass Company. [Page 12.]

[12] P. Mell and T. Grance, "The nist definition of cloud computing," Sep 2011. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/145/final [Page 12.]

[13] F. Richter, "Infographic: Amazon maintains cloud lead as microsoft edges closer," Feb 2024. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ [Page 13.]

[14] S. Al-Saqqa, S. Sawalha, and H. Abdel-Nabi, "Agile software development: Methodologies and trends," *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 14, p. 246, 07 2020. doi: 10.3991/ijim.v14i11.13269 [Pages 14 and 16.]

[15] H. Patil and G. Price, "Continuous software delivery explained - ericsson." [Online]. Available: https://www.ericsson.com/en/blog/2020/9/cicd-and-cdd-continuous-software-delivery-explained [Page 14.]

[16] "Unleashing the power of devops, agile, ci/cd, and devsecops: Why your business needs to embrace them now!" [Online]. Available: https://www.linkedin.com/pulse/unleashing-power-devops-agile-cicd-devsecops-why-your-business-kmvvc/ [Page 14.]

[17] E.-M. Schön, J. Thomaschewski, and M. J. Escalona, "Agile requirements engineering: A systematic literature review," *Computer Standards Interfaces*, vol. 49, pp. 79–91, 2017.

doi: https://doi.org/10.1016/j.csi.2016.08.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548916300708 [Page 15.]

[18] J. Beckman, "Agile statistics: How many companies use agile in 2023?" Aug 2023. [Online]. Available: https://techreport.com/statistics/how-many-companies-use-agile/ [Page 15.]

[19] dev2ops, "10+ deploys per day: Dev and ops cooperation at flickr," Jun 2009. [Online]. Available: http://dev2ops.org/2009/06/10-deploys-per-day-dev-and-ops-cooperation-at-flickr/ [Page 16.]

[20] S. Mezak, M. Ashley, N. Solomon, S. Sawyerr, K. Junčytė, and B. Washington, "The origins of devops: What's in a name?" Jul 2021. [Online]. Available: https://devops.com/the-origins-of-devops-whats-in-a-name/ [Page 16.]

[21] C. de Siebra, R. Lacerda, I. Cerqueira, J. P. Quintino, F. Florentin, F. Q. B. da Silva, and A. L. M. Santos, "From theory to practice: The challenges of a devops infrastructure as code implementation," in *International Conference on Software and Data Technologies*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52117190 [Page 17.]

[22] K. B. Gene Kim, George Spafford, *The Phoenix Project : A Novel about IT, DevOps, and Helping Your Business Win*. IT Revolution Press, 2015. ISBN 978-0988262508. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=4814cfff99c98dcaee1b7589bbda244b [Page 17.]

[23] J. Davis and K. Daniels, *Effective devops: Building a culture of collaboration, affinity, and tooling at scale*. O'Reilly, 2016. [Page 17.]

[24] P. Swartout, *Continuous Delivery and DevOps: A quickstart guide*. Packt. Publishing, 2012. [Page 17.]

[25] S. Das, "Devops lifecycle: Different phases in devops," Feb 2023. [Online]. Available: https://www.browserstack.com/guide/devops-lifecycle [Page 18.]

[26] "How does agile development differ from traditional software development methods." [Online]. Available: https://fastercapital.

com/topics/how-does-agile-development-differ-from-traditional-software-development-methods.html [Page 18.]

[27] T. Taylor, "Observability in devops is evolving beyond monitoring," Jan 2024. [Online]. Available: https://amazic.com/observability-in-devops-is-evolving-beyond-monitoring/ [Page 19.]

[28] R. Chandramouli, *Strategies for integration of software supply chain security in DevSecOps CI/CD pipelines*, 2023. doi: 10.6028/nist.sp.800-204d.ipd [Page 23.]

[29] M. Idris, I. Syarif, and I. Winarno, "Web application security education platform based on owasp api security project," *EMITTER International Journal of Engineering Technology*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:255799743 [Page 24.]

[30] "Owasp dependency-check." [Online]. Available: https://owasp.org/www-project-dependency-check/ [Page 24.]

[31] "Top ten owasp foundation." [Online]. Available: https://owasp.org/www-project-top-ten/ [Page 24.]

[32] "Sonarqube code analysis - documentation." [Online]. Available: https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/overview/ [Page 24.]

[33] b. Vijayan, A. Vijayan, Name, b. Wilson, B. Wilson, b. Lal, A. Lal, b. Warner, and J. Warner, "Trivy security scanner: Vulnerability scanning guide," Dec 2023. [Online]. Available: https://devopscube.com/trivy-security-scanner/ [Page 25.]

[34] "All-in-one open source security," Jan 2023. [Online]. Available: https://trivy.dev/ [Page 25.]

[35] S. I. Shamim, "Mitigating security attacks in kubernetes manifests for security best practices violation," *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:237205483 [Page 25.]

[36] "What is kubescape?" Feb 2024. [Online]. Available: https://www.armosec.io/kubescape/ [Page 25.]

[37] K. Research, "Infrastructure as code market," https://www.kbvresearch.com/infrastructure-as-code-market/, kBV Research. [Page 26.]

[38] O. Y. Mikhelson, "Infrastructure as code: Overview and application." [Online]. Available: https://api.semanticscholar.org/CorpusID:259671755 [Page 26.]

[39] A. Rahman, R. Mahdavi-hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, 12 2018. doi: 10.1016/j.infsof.2018.12.004 [Page 26.]

[40] *Infrastructure as Code Strategies and Benefits in Cloud Computing*. ProQuest Dissertations and Theses, 2022. [Online]. Available: https://scholarworks.waldenu.edu/dissertations/13265 [Page 26.]

[41] [Online]. Available: https://www.upguard.com/blog/declarative-vs-imperative-models-for-configuration-management [Page 28.]

[42] D. K. Sharma, "Terraform vs ansible: Key differences between devops tools," Mar 2024. [Online]. Available: https://k21academy.com/ansible/terraform-vs-ansible/ [Page 28.]

[43] K. MORRIS, *Infrastructure as code: Dynamic Systems for the cloud age*. O'REILLY MEDIA, 2021. ISBN 9781098114671 [Page 28.]

[44] "Pulumi - about." [Online]. Available: https://www.pulumi.com/about/ [Page 28.]

[45] Pulumi, "We are pulumi," Jun 2022. [Online]. Available: https://www.linkedin.com/pulse/we-pulumi-pulumi/ [Page 28.]

[46] E. ÖZDOĞAN, . CERAN, and M. T. ÜSTÜNDAĞ. [Online]. Available: https://dergipark.org.tr/en/download/article-file/3463145 [Page 29.]

[47] R. Joshi, "Configure a remote backend with aws s3 and dynamodb state locking in terraform," Oct 2023. [Online]. Available: https://dev.to/aws-builders/configure-a-remote-backend-with-aws-s3-and-dynamodb-state-locking-in-terraform-3jne [Page 29.]

[48] H. T. Co., "Container technology," Jan 1970. [Online]. Available: https://link.springer.com/chapter/10.1007/978-981-19-3026-3_7 [Page 31.]

[49] M. Moravík and M. Kontek, "Overview of docker container orchestration tools," *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pp. 475–480, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:232373731 [Page 31.]

[50] C. Carrión, "Kubernetes as a standard container orchestrator - a bibliometric analysis," *Journal of Grid Computing*, vol. 20, pp. 1–23, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254248844 [Page 33.]

[51] T. Nguyen Thanh, Y.-J. Yeom, T. Kim, D. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, p. 4621, 08 2020. doi: 10.3390/s20164621 [Page 33.]

[52] H. Agrawal, *Kubernetes Deep Dive*. Apress, November 2023, pp. 31–65. [Online]. Available: https://lens.org/122-135-046-373-630 [Page 33.]

[53] ——, *Manage Your Kubernetes Cluster Efficiently*. Apress, November 2023, pp. 293–347. [Online]. Available: https://lens.org/147-567-184-788-11X [Page 33.]

[54] Cloud Native Computing Foundation, "How kubernetes works," aug 2019. [Online]. Available: https://www.cncf.io/blog/2019/08/19/how-kubernetes-works/ [Page 33.]

[55] A. E. Nocentino and B. Weissman, *Kubernetes Architecture*. Berkeley, CA: Apress, 2021, pp. 53–70. ISBN 978-1-4842-7192-6. [Online]. Available: https://doi.org/10.1007/978-1-4842-7192-6_3 [Page 33.]

[56] [Online]. Available: https://spacelift.io/blog/kubernetes-sidecar-container [Page 34.]

[57] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm charts for kubernetes applications: Evolution, outdatedness and security risks," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023. doi: 10.1109/MSR59073.2023.00078 pp. 523–533. [Page 35.]

[58] F. Beetz and S. Harrer, "Gitops: The evolution of devops?" *IEEE Software*, vol. 39, no. 04, pp. 70–75, jul 2022. doi: 10.1109/MS.2021.3119106 [Pages 36 and 41.]

[59] A. Richardson, "Weaveworks is closing its doors and shutting down commercial operations.: Alexis richardson posted on the topic," Feb 2024. [Online]. Available: https://www.linkedin.com/posts/richardsonalexis_hi-everyone-i-am-very-sad-to-announce-activity-7160295096825860096-ZS67/?utm_source=thenewstack&utm_medium=website&utm_content=inline-mention&utm_campaign=platform [Page 36.]

[60] Fluxcd, "Fluxcd/flux2: Open and extensible continuous delivery solution for kubernetes. powered by gitops toolkit." [Online]. Available: https://github.com/fluxcd/flux2 [Page 37.]

[61] Argoproj, "Argoproj/argo-cd: Declarative continuous deployment for kubernetes." [Online]. Available: https://github.com/argoproj/argo-cd [Page 37.]

[62] "The practical guide to gitops," https://go.weave.works/rs/249-YDT-025/images/eBook_ThePracticalGuideToGitOps.pdf. [Page 38.]

[63] "Comparing docker hub and github container registry," Apr 2023. [Online]. Available: https://jfrog.com/devops-tools/article/comparing-docker-hub-and-github-container-registry/ [Page 42.]

[64] R. Šmíd. [Online]. Available: https://dspace.cvut.cz/bitstream/handle/10467/88177/F8-BP-2020-Smid-Radek-thesis.pdf?sequence=-1&isAllowed=y [Page 43.]

[65] M. Kotienieva, "VŠkp." [Online]. Available: https://vskp.vse.cz/89408_zavedeni-ci-cd-procesu-s-vyuzitim-gitops-ve-vybranem-podniku?order=type&page=1343 [Page 43.]

[66] M. D'AMORE. [Online]. Available: https://webthesis.biblio.polito.it/18142/1/tesi.pdf [Page 43.]

[67] W. Johansson, "A comparison of ci/cd tools on kubernetes," Jun 2022. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1676105/FULLTEXT01.pdf [Pages 43 and 44.]

[68] R. Alif and L. Munggaran, "Implementation of gitops in containerized infrastructure," *Rabit: Jurnal Teknologi dan Sistem Informasi Univrab*, vol. 9, no. 1, pp. 154–161, Feb. 2024. doi: 10.36341/rabit.v9i1.3787. [Online]. Available: https://jurnal.univrab.ac.id/index.php/rabit/article/view/3787 [Pages 43 and 44.]

[69] Ansible Community Documentation, “How to build your inventory - ansible community documentation,” https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html, March 2024. [Page 61.]

[70] K. Lamkin, “Cncf argo cd rollouts 2023 user survey results,” Jul 2023. [Online]. Available: https://blog.argoproj.io/cncf-argo-cd-rollouts-2023-user-survey-results-514aa21c21df [Page 62.]

Appendix A

Public Repositories

This appendix provides the source codes of AWS EKS deployments using Pulumi and Terraform, the established Helm chart for application deployment into EKS, Node.js application source code and the custom Python script to fetch the pipeline metrics from GitLab APIs. These source code repositories are discussed in Chapter 3.

A.1 IaC, Application and Helm Chart

A.1.1 Pulumi EKS Deployment

- GitHub Repository - Pulumi EKS Cluster Deployment

A.1.2 Terraform EKS Deployment

- GitHub Repository - Terraform EKS Cluster Deployment

A.1.3 Node.js Application Helm Chart

- GitHub Repository - Helm chart

A.1.4 Node.js Application

- GitHub Repository - Node.js Application

A.1.5 Custom Python Script for GitLab APIs

- GitHub Repository - GitLab APIs

Appendix B

Main equations

This appendix gives an explanation of equations that are used in Chapter 4.

When the sample sizes (n) are always ≥ 30, the Central Limit Theorem (CLT) can be applied. This theorem states that the distribution of a sample variable can be approximated to a normal distribution (i.e., a "bell curve") if the sample size becomes larger (n $\geq$ 30) regardless of the population's actual distribution shape [7]. As a result, Confidence Intervals (CI) are needed to quantify the uncertainty in the point estimate with a certain level of confidence (usually 95%).

B.1 Pooled estimate S_p of the common standard deviation

To calculate the Confidence Interval (CI), the pooled estimate S_p of the common standard deviation is calculated [9]. The samples from two distinct populations are independent if one has no connection with the other, according to the Central Limit Theorem (CLT). Therefore, the difference in the population means is the parameter of interest, i.e., $\mu_1 - \mu_2$.

The point estimate for the difference in population means is the difference in sample means:

$$(\bar{X}_1 - \bar{X}_2) \tag{B.1}$$

Where $\bar{X}_1$ and $\bar{X}_2$ denote the mean of the first and second samples.

The pooled estimate S_p of the common standard deviation is calculated as [9]:

$$S_p = \sqrt{\frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}} \tag{B.2}$$

Where S_1 and S_2 are the sample standard deviation of the first group and second group, and n_1 and n_2 are the sample sizes of the first and second group.

B.2 Confidence Interval (CI)

The Confidence Interval (CI) for a difference between two means, when both n_1 and n_2 are greater than 30, is given by [9] [10]:

$$C.I = (\bar{X}_1 - \bar{X}_2) \pm ZS_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}} \tag{B.3}$$

Where the critical value (Z-Value) is ≈ 1.96 for 95% confidence.

All the above formulas are valid when the two populations have similar variances. As a guideline, the ratio of the sample variances, $\frac{s_1^2}{s_2^2}$, must be between 0.5 and 2.

TRITA-EECS-EX-2024:492
Stockholm, Sweden 2024