Technology
Arts Sciences
TH Köln

Fraunhofer
IPT

Performance Optimization of Cloud-Native CI/CD Pipelines: A Comparative Study of GitOps and Traditional Approaches

RESEARCH PROJECT

by

Md Hazrat Ali

submitted as a requirement for

MASTER OF SCIENCE (M.SC.)

at

TH KÖLN UNIVERSITY OF APPLIED SCIENCES
INSTITUTE OF COMPUTER AND COMMUNICATION TECHNOLOGY

Course of Studies

COMMUNICATION SYSTEMS AND NETWORKS

First supervisor: Prof. Andreas Grebe
TH Köln University of Applied Sciences

Second supervisor: Talib SANKAL
Fraunhofer IPT

Cologne, February 2025

Contact details: Md Hazrat ALI
Grüngürtelstr. 120
50996 Köln
md._hazrat.ali@smail.th-koeln.de

Prof. Dr. Andreas GREBE
Cologne University of Applied Sciences
Institute of Computer and Communication Technology
Betzdorfer Straße 2
50679 Köln
andreas.grebe@th-koeln.de

Talib SANKAL
Fraunhofer IPT
Steinbachstraße 17
52074 Aachen
talib.sankal@ipt.fraunhofer.de

Abstract

In today's dynamic software development environment, optimizing cloud-native Continuous Integration and Continuous Deployment (CI/CD) pipelines are critical for achieving superior performance and enhancing reliability. This study aims to improve the performance of cloud-native CI/CD pipelines by comparing GitOps-based and traditional techniques. The goal is to assess the efficiency, scalability, and operational benefits of GitOps approaches compared to conventional CI/CD practices.

The research develops and implements two distinct CI/CD pipelines: one utilizing GitOps with Argo CD and the other employing conventional tools such as Jenkins. Both pipelines are organized within Kubernetes clusters, leveraging Prometheus and Grafana for performance tracking and to verify optimal functioning. Comparative evaluations examine important performance metrics such as system setup time, resource usage, fault tolerance, and operational overhead.

Additional resources, such as the grafana dashboard, are included for real-time monitoring to enhance analytical capabilities. Preliminary findings indicate that GitOps, by treating Git repositories as the single source of truth and leveraging declarative configurations, demonstrates superior scalability and consistency in application deployment.

This research enhances our understanding of cloud-native CI/CD practices by illustrating the differences between GitOps and traditional approaches across various development environments. Additionally, it provides practical advice for applying these strategies. The results offer clear recommendations for companies seeking to improve their CI/CD processes and adopt current DevOps methodologies.

Keywords: GitOps, CI/CD, Cloud-Native, Performance Optimization, Kubernetes, Prometheus, Grafana, Helm, Argo CD.

Acknowledgements

I express my deepest gratitude to the individuals and organizations whose invaluable support and contributions have successfully completed this research.

First and foremost, I extend my most profound appreciation to Mr. Talib Sankal, my research supervisor at Fraunhofer IPT, for allowing me to carry out this research. His guidance and support throughout the entire research process have been invaluable. The opportunity to collaborate with Fraunhofer IPT provided a practical and enriching experience that significantly enhanced the quality of this research.

I am also grateful to Prof. Dr. Andreas Grebe from the Institute of Computer and Communication Technology at Cologne University of Applied Sciences. His insightful feedback and academic guidance played a crucial role in shaping the direction of this research.

Special thanks go to Fraunhofer IPT for providing access to their OpenStack platform. This collaboration facilitated experiments and data collection, enriching the research's depth and applicability.

Lastly, I want to thank my family and friends for their unwavering encouragement and understanding during this demanding academic journey.

This research would not have been possible without the collective support of these individuals and organizations. Thank you for being a part of this academic endeavor.

Contents

List of Figures

List of Tables

List of Abbreviations

VM	**V**irtual **M**achine
CI	**C**ontinuous **I**ntegration
CD	**C**ontinuous **D**elivery
AWS	**A**mazon **W**eb **S**ervices
IaC	**I**nfrastructure as **C**ode

Chapter 1

Introduction

In the rapidly advancing field of software development, cloud-native technologies have become essential for enabling fast, scalable, and reliable software delivery. Continuous Integration and Continuous Deployment (CI/CD) pipelines are at the heart of this transformation, automating the process from code commits to production deployments. As organizations adopt cloud-native approaches, optimizing CI/CD pipelines for performance, scalability, and resource efficiency has become increasingly critical. These optimizations accelerate time-to-market and ensure that applications can scale effectively and are resilient in the face of growing demand [1].

This research project, "Performance Optimization of Cloud-Native CI/CD Pipelines: A Comparative Analysis of GitOps and Traditional Approaches," focuses on developing CI/CD pipelines at Fraunhofer IPT, where effective software delivery is crucial. The study compares two predominant approaches: GitOps-based CI/CD and traditional event-driven pipelines. Traditional CI/CD pipelines, in this case, will be implemented using Jenkins, a widely used automation server that facilitates building, testing, and deploying applications through event-based triggers. While Jenkins-based pipelines are effective, they often require manual intervention for infrastructure management and can be less automated in handling cloud-native deployments compared to GitOps approaches [2].

In contrast, GitOps-based CI/CD pipelines utilize Git as the single source of truth for both application and infrastructure configurations. GitOps improves automation, consistency, and reliability in the deployment process by automatically syncing changes in the Git repository with deployment environments using tools like GitLab, ArgoCD, and Kubernetes [3]. It also makes rollbacks easier to use.

This study looks at key performance indicators (KPIs) like deployment speed, resource utilization, and error rates to find out how to make cloud-native CI/CD pipelines work better in production environments. The project will leverage cutting-edge tools, including Kubernetes for container orchestration, GitLab for CI/CD automation, ArgoCD for GitOps deployments, Helm for package management, and monitoring tools like Prometheus and Grafana for real-time performance tracking. This study aims to assess the effectiveness of GitOps and Jenkins-based traditional CI/CD pipelines, helping organizations optimize their pipelines based on performance data [4].

Despite the growing popularity of both GitOps and traditional CI/CD pipelines, there remains a lack of comprehensive comparative studies assessing their performance in terms of key metrics. In the end, this research will make a framework

that helps Fraunhofer IPT and the software development community as a whole by making cloud-native CI/CD practices more reliable and efficient.

The goal of the study is to help improve CI/CD pipeline methods for better performance in cloud-native environments [5] by finding best practices and optimization strategies.

Chapter 2

Overview of CI/CD and GitOps

2.1 Historical background of CI/CD

CI/CD is a software development methodology that integrates continuous integration and continuous delivery. In other words, it is a method of automating the deployment of code modifications as they are implemented. CI/CD is a collection of practices that assist developers in rapidly developing and testing their applications, thereby minimizing risk and enhancing quality [6]. Additionally, it allows them to implement rapid modifications without disrupting production environments. The idea behind CI/CD is simple: Developers build code in batches, which are frequently referred to as "pipelines." Subsequently, they execute tests on each batch before combining the results into a main branch for deployment. The primary objective is to eliminate human errors by automating repetitive tasks, such as evaluating new builds on staging or QA environments before deploying them into production. Conseuently, developers may dedicate more time to the development of code rather than the management of infrastructure issues, such as the deployment of new versions of applications or the resolution of errors that are the result of outdated configurations or failed dependencies between various services. This helps to ensure high-quality releases at a large scale while saving time and money on manual processes like deployments every night or every week due to outages caused by human error and malicious attacks.

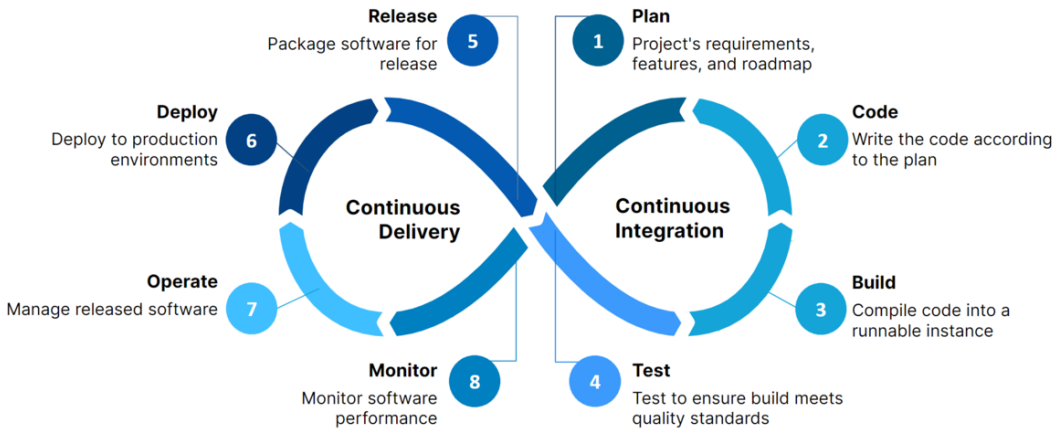

FIGURE 2.1: CI/CD Reference Architecture [7]

2.1.1 Continuous Integration (CI)

Continuous Integration (CI) is a software development methodology that involves the consistent testing and merging of code to ensure that it satisfies specific standards, including security, performance, and quality. The concept has been around for an extended period and has made significant advancements over time. The term "Continuous Integration" was first used by Grady Booch in his 1991 book Object-Oriented Design with Applications [8]. Since then, it has become an important part of building software because it makes combining and testing code more manageable. It is the process of autonomously creating and testing the code whenever developers save changes in tools such as Subversion or Git [9]. This ensures that all tests are successful before the code is shared or utilized, thereby allowing developers to be confident that their work is prepared for use.

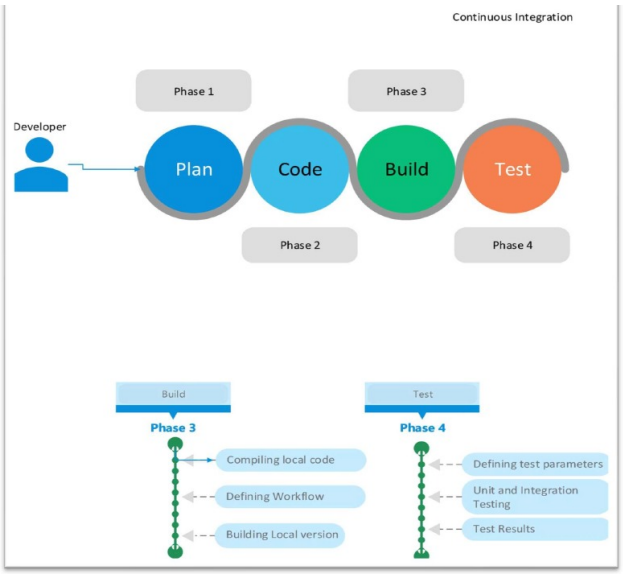

FIGURE 2.2: CI Workflow [10]

The CI cycle has four main steps: planning, coding, building, and testing [11]. While all four steps are important, the building and testing steps are the most critical because they focus on automating the work. This automation makes the process faster and more reliable for developers.

These phases can be defined as follows: -

Plan: The planning phase is the first step of a project. It includes all the necessary steps to define and prepare for implementing an idea or a product.

Code: This phase is more focused on core coding/developing tasks. It includes making major decisions like deciding languages, frameworks to be used, etc.

Build: The build phase consists of actions like code compiling, defining workflow, and building local code versions, making code ready for the testing phase.

Test: In the testing phase, integration of different developers' work is done after running the local builds through an automated testing pipeline. Which is accompanied by feedback and reviews on the work done.

These steps are defined differently from project to project, but the overall structure remains the same.

2.1.2 Continuous Delivery (CD)

Continuous Deployment or Continuous Delivery (CD) is a way of delivering software that uses automated and reliable tools and processes. The goal is to deliver value to customers faster by constantly improving the software's quality through automation and better processes [13]. With CD, developers can release updates without waiting for manual code reviews. This approach allows for quick and frequent customers feedback, which helps create better software faster. Every change goes through automated testing before being released, reducing the chances of bugs or issues during deployment and minimizing downtime [12]. Since updates happen often, this process becomes a core part of ongoing development.

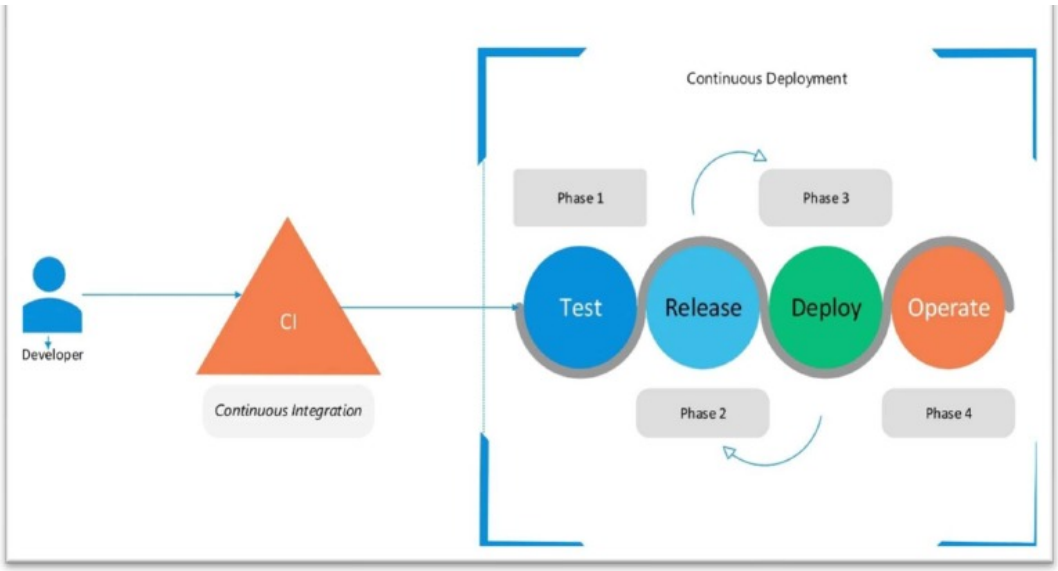

FIGURE 2.3: CD Workflow [10]

The CD process includes four main phases: test, release, deploy, and operate [14]. These phases are connected and occur in sequence. First, Continuous Integration (CI) creates a build that can be deployed, which is then followed by CD methods.

The four phases are:

Test: The testing phase involves running various tests, from unit to integration tests. This process ensures high code quality and results in a working product version.

Release: During the release phase, changes are documented, and the product version is prepared for release.

Deploy: The released product is deployed to a hosting server, making it available for use by the end-user.

Operate: Once the code is live, it is monitored continuously, and performance records are kept. These insights help decide future updates and improvements.

2.1.3 CI/CD Pipeline

CI/CD Pipeline In software engineering, a pipeline includes a sequence of processing steps (such as processes, threads, coroutines, or functions), organized so that the output of one step becomes the input of the next [15]. Pipeline in CI/CD: The

CI/CD pipeline is a series of automated processes used to manage and deploy software. When done correctly, it reduces human errors and improves feedback loops during the SDLC, enabling teams to deliver smaller updates more frequently [16].

To implement the pipeline, a YML-based script is created and added. This script contains all the required details about the runtime environment and the actions to be executed.

Figure 3 shows the structure of the CI/CD pipeline.

The CI/CD pipeline has four main phases: Commit, Build, Test, and Deploy [15].

Commit

Developers select a version control system like GitHub or GitLab to organize and structure the codebase. Depending on the project, developers carry out development either individually or collaboratively [15]. The pipeline is triggered by actions such as pull requests (downloading code from the version control system to a local machine) or push requests (uploading code changes to the version control system).

Build

Once the pipeline starts, the YML script is added to the version control system. This script automatically sets up the virtual environment and executes the build for the project.

Test

After the build is complete, the software undergoes predefined tests written by the developers. These tests ensure the quality of the product before moving to the next phase.

Deploy

Once all tests successfully pass, the product is deployed and made available for use.

2.2 Models of CI/CD

2.2.1 Cloud-Native CI/CD Architectures

Cloud-native CI/CD (Continuous Integration/Continuous Delivery) is a modern approach to developing and deploying applications designed specifically for cloud environments. It takes advantage of cloud-native principles such as scalability, flexibility, and automation to streamline software delivery [17]. Unlike traditional systems, cloud-native CI/CD allows developers to work with modular components, ensuring faster updates and more efficient resource use. Cloud-native technologies enable organizations to build and run scalable applications across diverse environments, including public, private, and hybrid clouds [18]. This approach incorporates key elements like containers, microservices, service meshes, immutable infrastructure, and declarative APIs, which help drive flexibility and efficiency. Key technologies in cloud-native CI/CD include microservices, containers, and orchestration tools. Microservices break applications into smaller, independent units, allowing teams to work on and deploy updates for specific components without affecting the entire system. Containers, such as those created with Docker, package these microservices with their dependencies, ensuring consistency across environments. Orchestration platforms like Kubernetes further simplify managing these containers by automating deployment, scaling, and health monitoring [18].

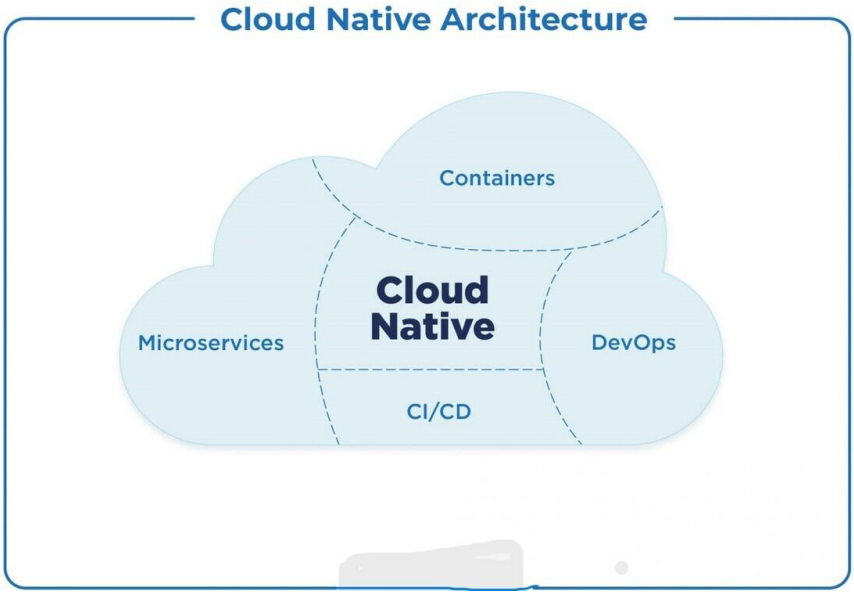

FIGURE 2.4: Cloud Native Architecture [18]

The adoption of cloud-native CI/CD has transformed software delivery by promoting continuous delivery and eliminating manual bottlenecks. Developers can now automate most stages of the pipeline, reducing errors and accelerating deployment cycles. This shift also integrates development and operations workflows, enhancing collaboration and aligning with modern agile practices. By combining automation, modularity, and scalability, cloud-native CI/CD helps organizations deliver high-quality software more efficiently while adapting to the ever-changing demands of the industry [17].

2.2.2 Event-Driven CI/CD (Traditional Approach)

Event-driven CI/CD pipelines are a common approach used in software development to automate the process of building, testing, and deploying applications [19]. These pipelines trigger specific actions based on specific events, such as when a developer commits code or opens a pull request. Once the event occurs, the pipeline automatically starts tasks like compiling code, running tests, and deploying the application. This system ensures that each change to the codebase is processed and deployed quickly, allowing developers to detect issues early and keep software up to date.

Tools like Jenkins, GitLab CI [20], and CircleCI are commonly used in traditional event-driven pipelines. These tools automate various stages of the development process, making it easier for developers to manage their workflows. Jenkins, for example, is widely used due to its flexibility, allowing teams to set up triggers for builds, tests, and deployments. However, these tools can require significant configuration and may need manual intervention to handle specific tasks, such as infrastructure management and scaling, which can be time-consuming and error-prone.

Despite their strengths, traditional event-driven CI/CD methodologies have several challenges. One of the most prominent limitations is the difficulty in scaling these pipelines, especially in large, complex projects. As applications grow, the pipelines need to handle more triggers and processes, which can lead to slow execution times and increased complexity. Additionally, while event-driven pipelines offer automation, they often still require manual intervention for infrastructure management, such as scaling the infrastructure or troubleshooting issues that arise during deployment. This reliance on human intervention can undermine the benefits of automation and increase the risk of errors or delays in the deployment process.

Thus, traditional event-driven pipelines may struggle to keep up with the speed and flexibility required in modern development environments.

2.2.3 Declarative CI/CD Models (GitOps Approach)

GitOps, introduced by Weaveworks in 2017, is a methodology that uses Git as the single source of truth to manage cloud-native applications and infrastructure. It defines the desired state of applications and infrastructure declaratively in Git repositories, focusing on what the system should look like rather than how to implement it. Git tracks and manages configuration changes, ensuring consistency, reducing errors, and simplifying rollbacks [22, 24, 25] through its version control features. GitOps integrates seamlessly with Kubernetes, enhancing continuous delivery and automation.

A significant advantage of GitOps is its developer-friendly design, as developers are al- ready familiar with Git workflows. They make changes in Git, submit pull requests, and merge them after review. Automated tools such as ArgoCD, Flux, and Helm synchronize the system's state with the desired state defined in the repository. These tools automatically apply updates to Kubernetes clusters and facilitate rollbacks if necessary, streamlining deployment and minimizing misconfigurations. GitOps supports two configuration reconciliation methods: push-based [21], which uses CI/CD pipelines to trigger updates, and pull-based, where an operator in the environment continuously syncs changes from the repository. This ensures that the desired state is always maintained. GitOps also includes monitoring and observ-

ability tools that can find and fix any drift between the desired and live states. This makes managing cloud-native apps more predictable and reliable.

2.3 Foundations of GitOps

GitOps is a modern operational model where Git repositories serve as the single source of truth for both infrastructure and application configurations. Git commits make changes, and tools like ArgoCD or Flux automatically apply these changes to the target environment [1] [2]. This method makes it easier to manage cloud systems, helps teams work together, and reduces errors, especially in Kubernetes environments [3].

GitOps relies on three main ideas: declarative configuration, version control, and pull-based deployment. Declarative configuration means clearly defining the system states in Git, ensuring consistency. Version control tracks all changes, making audits and rollbacks easy [4]. The pull-based deployment model allows tools like ArgoCD to watch for updates in Git and apply changes continuously, keeping the system synchronized [5].

Compared to traditional CI/CD pipelines, GitOps offers advantages. Traditional workflows push changes to the environment after successful builds and tests. GitOps continuously synchronizes the environment with the configurations in Git, ensuring consistency. This results in better auditing, transparency, and scalability, making GitOps more efficient and reliable, especially for complex Kubernetes environments [3][4].

Chapter 3

Technical Background

The following is a list of existing tools used in this research project.

3.1 Git/GitHub

Modern software development relies on Git and GitHub for version control and collaborative coding. The distributed version control system Git, created by Linus Torvalds in 2005 [27], enables developers to track changes in source code during software development. It simplifies branching, merging, and tracking changes in huge codebases. Because Git is distributed, each developer has a complete copy of the repository, enabling offline work and decreasing reliance on a central server. Based on Git, GitHub offers a web-based interface and cloud-based hosting for faster collaboration and additional features such as pull requests, issue tracking, and continuous integration. Git and GitHub [27] enable team collaboration without overwriting contributions. Using branching and pull requests, developers can offer modifications that are vetted and incorporated into the main project. GitHub promotes creativity and teamwork by sharing and showcasing projects to the global coding community. Combining with GitHub Actions for automation and code scanning for security creates a complete platform for development and deployment. Development tools like Git and GitHub have transformed code management and collaboration for developers, becoming vital in software engineering [26].

3.2 Gitlab

GitLab is a web-based platform designed to support DevOps practices by combining tools for collaborative software development, version control, and automated CI/CD workflows. Created in 2011 by Dmitriy Zaporozhets and Valery Sizov [29], the platform prioritizes privacy and self-hosting, distinguishing itself from similar services like GitHub. One of GitLab's primary innovations is its built-in CI/CD pipelines [28], which automate critical development processes such as testing, deployment, and monitoring, eliminating the need for external tools. The platform also incorporates Agile project management features, including issue tracking, milestone scheduling, and progress analytics, enabling teams to oversee the entire software development lifecycle in a centralized environment. By integrating these tools into a single system, GitLab minimizes disjointed workflows, promotes collaboration among developers and operations teams, and streamlines workflows to improve efficiency and project oversight.

3.3 Docker

Docker is an open-source platform designed to make developing, shipping, and running applications more efficient. It uses containerization, where applications and their dependencies are packaged into portable containers that work consistently in development, testing, and production environments. Released in 2013 by Solomon Hykes, Docker has transformed software workflows by enabling seamless portability and scalability. Its lightweight containers share the host system's kernel, making them faster and more efficient than traditional virtual machines [30].

Docker is based on a client-server architecture, enabling efficient container management and deployment. The Docker client communicates with the Docker daemon, which builds, runs, and manages containers. The client and daemon can run on the same machine or interact remotely using the REST API. The Docker host includes the daemon, images, containers, and storage. The client pulls container images from a Docker Registry, like Docker Hub, for container creation [31].

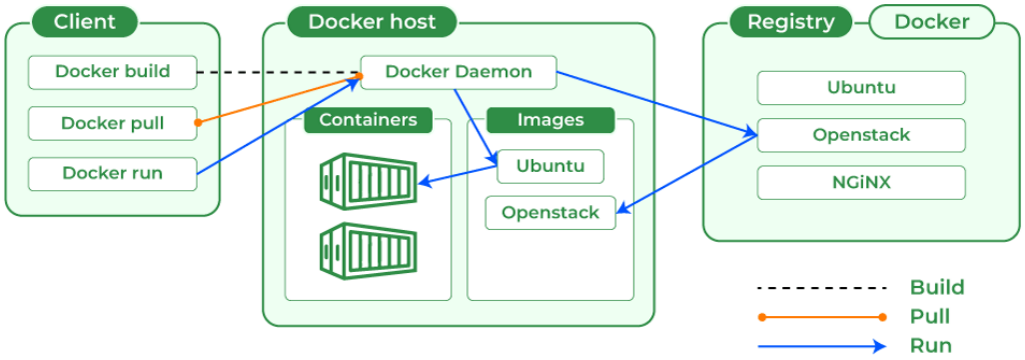

FIGURE 3.1: Docker Architecture [31]

Key Docker Components:-

- **Docker Images:** An image includes instructions for building a Docker container. It is simply a read-only template. It is used for storing and shipping applications. Images are a crucial element of the Docker experience because they allow developers to collaborate in new ways that were previously impossible.
- **Docker Containers:** Containers are built from Docker images, which are prebuilt applications. We can use the Docker API or CLI to start, stop, delete, or relocate containers. A container can only access the resources defined in the image unless additional access is specified in the container when generating the image.
- **Docker Volumes:** Docker containers mount Docker Volumes file systems to keep the state of containers (data) created by the running container. The volumes are stored on the host regardless of how long the container runs. This allows users to transfer file systems between containers and back data quickly.
- **Docker Registry:** The Docker Hub is a public registry where Docker images are stored. We can also manage a private registry. We can retrieve the necessary images from our configured registry by using the docker run or docker

pull commands. The docker push command pushes images to the configured registry.

- **Docker Networking:** Docker networks enable full isolation for Docker containers. This means that a user can connect a Docker container to multiple networks while requiring only a few operating system instances to run the workload.
- **Dockerfile:** A script that contains instructions for building a Docker image. It specifies the basic image, application code, dependencies, and behavior of the container.

3.4 Kubernetes

Kubernetes is a portable, adaptable, open-source platform for managing containerized workloads and services that allows for declarative setup and automation.

Kubernetes was originally developed by Google as an internal container management project called Borg before being open-source on June 7, 2014. In July 2015, it was donated to the Cloud Native Computing Foundation (CNCF) [32].

The Architecture of Kubernetes Kubernetes follows a cluster-based architecture comprising two main components: the control-plane (master node) and worker nodes. These work together to manage and execute containerized workloads.

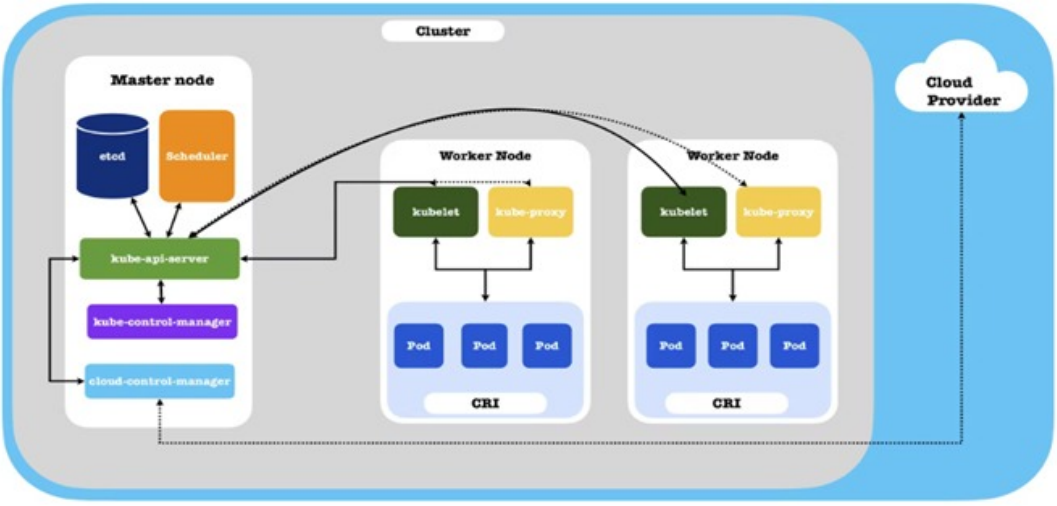

FIGURE 3.2: Kubernetes Architecture [33]

Master Node Components:

- **kube-apiserver:** Acts as the interface for all cluster operations, exposing the Kubernetes API for user and system interactions.
- **etcd:** A distributed key value store that stores all cluster data, ensuring consistency and availability.
- **kube-scheduler:** Assigns pods to appropriate worker nodes based on resource availability and policies.
- **kube-controller-manager:** Ensures the cluster remains in the desired state, handling tasks like replication and endpoint management.

- **kube-controller-manager:** Integrates with cloud platforms for managing resources such as nodes, storage, and load balancers [33].
- **cloud-controller-manager:** Integrates with cloud platforms for managing resources like nodes, storage, and load balancers

Worker Node Components:

Worker nodes execute workloads and host containerized applications. Their main components include:

- **Kubelet:** An agent that communicates with the control plane to manage pod operations on the node.
- **kube-proxy:** A networking component that handles communication between services and pods, ensuring smooth traffic routing.
- **Container Runtime:** Container Runtime: Executes containers, such as Docker, containerd [34].

Core Kubernetes Objects:-

- **Pods** The smallest deployable units in Kubernetes, containing one or more containers that share networking and storage.
- **Services:** Provide stable networking to expose applications running in pods.
- **Persistent Volumes (PV) and Persistent Volume Claims (PVC):** Enable storage that persists beyond the lifecycle of pods, essential for stateful applications.
- **ConfigMaps and Secrets:** Store configuration data and sensitive information, allowing separation from application code.

Kubernetes Deployment and Service Files:-

YAML is a human-readable data-serialization language. Applications and configuration files commonly use it to store or transmit data. In this context, Kubernetes uses YAML files to define application configurations. Two essential files are the deployment and service files.

3.5 Helm

Helm is a widely used package manager for Kubernetes that simplifies application deployment and management. It uses pre-configured templates called "charts" to describe Kubernetes resources, enabling consistent deployments across environments. Helm automates complex configuration tasks, reducing manual errors and saving time during deployment processes [35]. One of Helm's key features is version control, which allows users to roll back to previous states if issues arise, making it highly reliable for managing applications in dynamic Kubernetes environments [36].

3.6 Jenkins

Jenkins is an open-source automation server designed for continuous integration (CI) and continuous delivery (CD). It streamlines the software development process by automating tasks such as building, testing, and deploying applications [37]. Jenkins offers a vast library of plugins, enabling integration with tools like Kubernetes

and Docker, which enhances its flexibility and adaptability in modern workflows. Developers can define pipelines as code in Jenkins, ensuring consistent and repeatable workflows that improve reliability and efficiency in software delivery [38].

3.7 ArgoCD

ArgoCD is a declarative GitOps continuous delivery tool tailored for Kubernetes environments. It automates application deployments by ensuring that the live environment matches the desired state stored in Git repositories [39]. With its support for Git-based workflows, ArgoCD offers version control, auditability, and easy rollback capabilities, making it a valuable tool for managing Kubernetes applications. The tool also integrates seamlessly with Helm charts and provides a user-friendly web interface for monitoring and managing deployments effectively [40].

3.8 Prometheus

Prometheus is an open-source monitoring tool that collects time-series data from various systems and services. It efficiently handles real-time data and allows users to query it using PromQL, a flexible query language. Prometheus uses a multi-dimensional data model, where time-series data is identified by metric names and key-value pairs. It follows a pull-based model over HTTP for data collection but also supports pushing data through an intermediary gateway. It operates independently without relying on distributed storage, and its targets can be discovered dynamically through service discovery or set manually. Prometheus integrates well with Kubernetes and CI/CD tools for automated monitoring and provides multiple options for graphing and dashboards. Its built-in alerting system, AlertManager, helps teams set up notifications to quickly respond to issues and minimize downtime [41].

3.9 Grafana

Grafana is a widely used open-source tool for visualizing data from different sources like Prometheus, Elasticsearch, and InfluxDB. It provides customizable dashboards that allow users to easily monitor system performance and detect problems. Grafana's simple interface makes it easy to create interactive and dynamic visualizations. It also supports advanced alerting, sending notifications through email or other communication tools when predefined conditions are met. Together with Prometheus, Grafana provides a complete solution for monitoring and visualizing data in cloud-native environments [42].

Chapter 4

Comparative Study of CI/CD Tools

4.1 Traditional CI/CD Approach

Traditional CI/CD, typically associated with a push-based pipeline, starts with the Continuous Integration (CI) steps and progresses through a series of scripts or manual commands, such as kubectl, to push changes to the Kubernetes (k8s) cluster. Many CI/CD environments have widely adopted this method. However, this approach has several key drawbacks. It lacks a declarative deployment process, which can result in inconsistencies when managing the desired state. Lastly, the live state of the cluster doesn't always match the desired state because this method doesn't use modern Kubernetes operators like ArgoCD or Flux, which make sure that the states are always in sync.

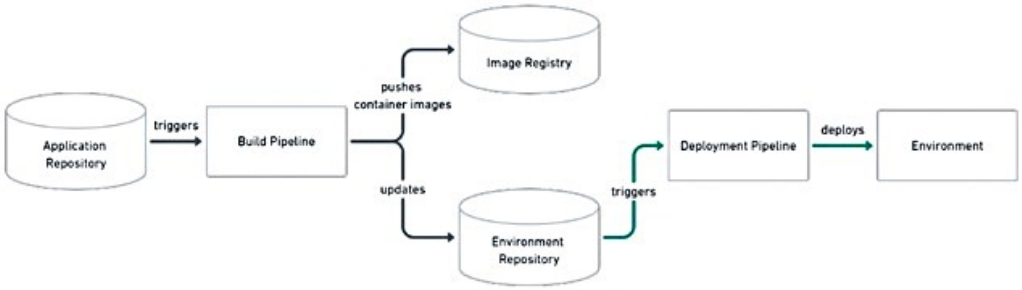

FIGURE 4.1: Push-based pipeline [43]

The following section describes of CI/CD tools in traditional approaches.

4.1.1 Bamboo

Bamboo is a commercial Continuous Integration (CI) and Continuous Deployment (CD) tool developed by Atlassian. It supports various technologies like AWS CodeDeploy, Docker, Maven, Git, and SVN, making it versatile for diverse ecosystems [44]. Bamboo detects new branches in repositories and applies the main CI configuration automatically, simplifying workflow management. It integrates seamlessly with Atlassian tools like Jira, enhancing project tracking and management [46]. The tool offers efficient deployment with built-in support for Docker agents and AWS tasks, along with customizable notifications through email or pop-ups. While it is free for open-source projects, non-open-source users get only a seven-day trial. Bamboo is well-documented and provides strong support, but its interface can be less intuitive, and the setup process may be challenging for new users. Despite these challenges, Bamboo remains a reliable option for traditional CI/CD pipelines [45].

4.1.2 CircleCI

CircleCI is a cloud-based Continuous Integration (CI) and Continuous Deployment (CD) tool known for its simplicity, speed, and flexibility [47]. It supports various environments, including web, mobile, and containerized applications, making it suitable for a wide range of development workflows. CircleCI integrates seamlessly with popular tools like GitHub, Heroku, and AWS, streamlining deployment processes. With features like automatic parallelization, smart notifications, and customizable settings, CircleCI simplifies complex CI/CD workflows while improving efficiency and reducing build times. The tool is praised for its clean user interface, fast support, and quick setup, often taking as little as 20 seconds to configure [48]. Users benefit from robust support for debugging, timely notifications, and strong documentation. However, there are some drawbacks, such as occasional compatibility issues after upgrades and the need for extra configurations in specific scenarios. Despite these minor limitations, CircleCI remains a popular choice for traditional CI/CD pipelines due to its ease of use and reliable performance.

4.1.3 Codeship

Codeship is a continuous integration and deployment tool designed for web applications, including PHP, Rails, Java, Python, and Go. It simplifies the process of running automated tests and setting up deployment pipelines after changes are made in a repository. This helps developers test and release frequently, ensuring fast feedback for improving software development [50]. Codeship integrates with popular platforms like GitHub, Bitbucket, Slack, Heroku, Amazon EC2, and HipChat. Its main features include easy deployment, simple setup, and intelligent notifications to keep teams informed [50]. However, Codeship has some drawbacks. Some users have mentioned that it lacks support for storing build artifacts. There are also occasional issues with notifications, and many users feel that the documentation is insufficient compared to tools like Jenkins. Additionally, some users have pointed out that Codeship's pricing can be high, and they would like to see more plugins added to extend its functionality [49]. Despite these issues, many users appreciate its simplicity, intuitive UI, and quick setup process, making it a good choice for smaller teams or those with limited budgets.

4.1.4 TeamCity

TeamCity is a versatile Continuous Integration (CI) server that is easy to set up and supports a variety of version control systems, testing, and deployment out of the box. It provides instant feedback on issues and test failures, allowing users to address errors without waiting for the full build to complete. TeamCity offers a flexible build configuration, with the ability to run tests across multiple platforms simultaneously. It also provides detailed reports on build success rates and code quality, along with excellent integration with version control systems such as Git and Mercurial. Additionally, it supports mixed authentication methods, making it suitable for diverse team setups [51].

While TeamCity is praised for its user-friendly interface and strong automation features, some users have found the setup and upgrade processes challenging. Documentation can be difficult to navigate, and certain terminology may require external research. Despite these issues, TeamCity's extensive customization options, ability to automate the development process, and scalability make it a good fit for teams of

all sizes and environments, though it may require more effort for initial setup and ongoing management [51].

4.1.5 Travis CI

Travis CI is a popular Continuous Integration (CI) tool that integrates seamlessly with GitHub, automating build, test, and deployment processes. Initially designed for Ruby on Rails, it now supports various programming languages and platforms. By adding a configuration file and setting up a webhook, Travis CI triggers builds automatically for every commit and pull request. Its free access to open-source projects and simple GitHub integration make it a user-friendly choice for developers [52].

However, Travis CI has some limitations. Users may face occasional stability issues, such as freezes during testing, and the .travis.yml configuration can be complex for beginners. Additionally, its limited support for Docker can pose challenges for containerized workflows. Despite these drawbacks, Travis CI remains a widely used tool, especially for open-source projects [53].

4.1.6 Jenkins

Jenkins is a widely used open-source tool for implementing Continuous Integration (CI) and Continuous Deployment (CD) processes. Originally, created by Kohsuke Kawaguchi under the name Hudson, Jenkins was later renamed in 2011 after a dispute with Oracle [56]. Jenkins stands out due to its cross-platform compatibility, running seamlessly on operating systems like Windows, Linux, and macOS, and its flexibility to support multiple programming languages and tools [55]. This adaptability makes Jenkins one of the most popular CI/CD tools in the software development industry, with a large and growing community of users. Jenkins offers a rich ecosystem of over 1200 plugins, allowing it to integrate with virtually any version control system, build tool, or deployment platform, including GitHub, SVN, and Maven [55]. Its web-based graphical interface simplifies configuration, enabling teams to set up workflows tailored to their project requirements with ease [55]. Jenkins supports both simple tasks like running unit tests and compiling code, as well as complex pipelines that involve multiple stages of testing, validation, and deployment [55]. It is used by major companies such as Facebook, Netflix, and LinkedIn, highlighting its versatility and reliability [54] While Jenkins is highly regarded for its extensibility and powerful features, it is not without its challenges. Users often mention that the interface is outdated and not always intuitive, requiring multiple clicks to access specific information. Additionally, configuring advanced plugins and managing updates can be time-consuming, as updates may occasionally lead to system issues. Despite these limitations, Jenkins remains a cornerstone of traditional CI/CD workflows due to its scalability and ability to automate repetitive tasks, which helps teams save valuable time and effort. For cloud-native environments, newer approaches like GitOps may offer more optimized performance and scalability, but Jenkins continues to excel in traditional CI/CD implementations.

Jenkins Vs Other Tools:

Table 4.1 compares several key features of Jenkins and five other popular CI/CD tools: Bamboo, CircleCI, Codeship, TeamCity, and Travis CI. A symbolic rating scale is used, ranging from - - - (very poor or not supported) to +++ (excellent or fully supported). These features capture critical aspects such as plugin ecosystems, scalability, and community support, essential for evaluating a CI/CD platform's suitability in various organizational contexts.

TABLE 4.1: Comparison of Traditional Tools

Feature	Jenkins	Bamboo	CircleCI	Codeship	TeamCity	Travis CI
Core Strength	+++	++	+	+	++	+
Plugin Power	+++	0	0	-	+	0
Customization Level	+++	+	+	0	++	+
Scalability for Projects	+++	++	++	0	++	0
Workflow Complexity	+++	+	+	-	++	0
Cost/Openness	+++ (free)	- (paid)	+ (mixed)	0	+ (mixed)	+ (mixed)
Community Support	+++	+	++	-	+	++

Explanation of Comparison Metrics (Table 4.1)

- **Core Strength:** Represents the foundational capabilities of the CI/CD tool, such as version control integration, build scheduling, and artifact management. Higher ratings indicate a robust, well-tested core system.
- **Plugin Power:** Evaluates the availability and variety of plugins or extensions. A higher rating (+++) signifies a rich ecosystem that allows extensive customization and integration with third-party services.
- **Container-Level Integration:** Assesses how well the tool handles container-based workflows (e.g., Docker, Kubernetes). Tools rated highly in this category often provide native container support or specialized plugins.
- **Scalability for Projects:** Focuses on the tool's ability to handle large codebases, multiple teams, and concurrent pipelines without performance issues. Higher ratings suggest better load handling and resource management.
- **Workflow Complexity:** Examines whether the tool can orchestrate advanced pipelines, including parallel stages, conditional logic, and external integrations. A +++ indicates strong support for complex build/deployment pipelines.
- **Cost:** Reflects the pricing model (open-source, freemium, or paid). +++ usually denotes free/open-source solutions, whereas lower ratings (- or 0) imply costs or limited free tiers.
- **Community Support:** Looks at the size and activity of the tool's user base, as well as the frequency of updates and availability of online resources (forums, documentation, etc.). A high rating means an active community and regular maintenance.

As shown in Table 4.1, Jenkins excels in multiple areas, particularly in its extensive ecosystem, plugin power, community support, and cost. Bamboo and TeamCity, on the other hand, are known for their strong core strength and workflow complexity support, which reassures users of their reliability. However, their reliance on paid licenses yields lower ratings in cost. CircleCI's cloud-first approach results in moderate ratings, emphasizing ease of setup over deep customization. Travis CI remains a popular hosted solution with mixed scores across most categories.

Why Jenkins?

Jenkins is widely adopted due to its extensibility, flexibility, and strong community support. It is an open-source solution, making it cost-effective for organizations of all sizes. Jenkins supports a wide range of programming languages and platforms, allowing it to adapt to various project requirements. Its ecosystem of over 1,200 plugins facilitates seamless integration with third-party tools and services, enabling advanced container-level integration and complex workflows.

As indicated in Table 4.1, Jenkins achieves top (+++) ratings in plugin power, community support, and cost, reflecting its status as a robust and versatile choice for CI/CD pipelines. Its active community also ensures frequent updates and ample resources for troubleshooting and learning.

4.2 GitOps Approach

In a pull-based GitOps pipeline, CI remains unchanged from a push-based approach, but CD differs: operators such as ArgoCD, Flux, Helm Operator, WKSctl, Gimlet, and Fleet continually compare the live Kubernetes cluster to the desired state stored in Git, updating whenever discrepancies arise [21]. Unlike push-based deployments that only act on repository changes, pull-based workflows maintain constant alignment and revert any unauthorized alterations. Tools like Helm Operator specialize in Helm-based applications, WKSctl manages cluster configurations, Gimlet provides a developer-friendly platform, and Fleet scales GitOps to large multi-cluster environments. Nevertheless, ArgoCD and Flux remain the most widely adopted operators for implementing a pull-based GitOps strategy [21]

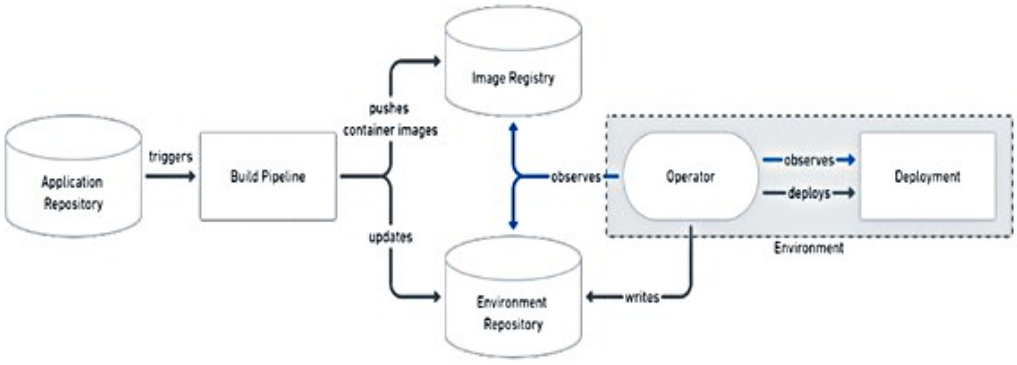

FIGURE 4.2: Pull-based pipeline[43]

4.2.1 Flux

Flux is an open-source tool that keeps Kubernetes applications synchronized with the configurations stored in a repository [38, 39]. It operates through a set of GitOps toolkit components—installable either directly within the target cluster or in a dedicated cluster—which include Kubernetes controllers and custom resource definitions (CRDs). Flux is configured declaratively using CRDs, meaning that configuration details are defined in manifest files. These files specify repository settings or the components to be deployed, with dedicated resources for each repository type (for example, GitRepository for Git-based repositories and HelmRepository for Helm-based ones). The tool also supports configuration management tools like Kustomize and Helm; specifically, the HelmRelease resource manages Helm application deployments [38]. At the core of Flux is its source controller, which continually monitors repositories for changes. When it detects an update—such as a new commit in a Git repository—the controller retrieves the changes and generates artifacts that other controllers use to update the application state. After deploying the resources, Flux monitors the reconciliation process to ensure everything remains healthy. Additionally, developers can manage the deployment sequence by establishing dependencies between resources, allowing customized deployment workflows [38].

4.2.2 ArgoCD

ArgoCD is an open-source tool that follows a pull-based GitOps approach, checking for updates every three minutes, similar to Flux [58, 60]. However, the key differences between ArgoCD and Flux lie in how they are configured and the extra features ArgoCD offers. ArgoCD uses a controller to monitor custom and labeled resources and remote repositories. The controller can be installed in a separate cluster. Unlike Flux, which relies on custom resource definitions (CRDs), ArgoCD uses native Kubernetes resources like Secrets and ConfigMaps. It also introduces custom resources such as Projects, Applications, and ApplicationSets—where Projects group applications together, and Applications represent individual components. For deploying Helm charts, ArgoCD uses the Application resource instead of Flux's HelmRelease resource [60]. ArgoCD manages deployments in three phases: pre-sync, sync, and post-sync, which provide more flexibility than Flux. Within each phase, resources are deployed in waves, with the order determined by resource type and name. The next wave is only triggered once the current one is confirmed to be healthy [57, 60]. ArgoCD also stands out because of its web interface, which makes it easier to configure, debug, and monitor applications—especially during initial development. Additionally, it offers features like selective reconciliation, sync windows for syncing specific parts of an application on demand, and built-in user management [57, 60]. These extra capabilities, including the web UI and selective synchronization, made ArgoCD the preferred choice for this project.

Argocd Vs Other Tools:

Table 4.2 compares six GitOps tools—ArgoCD, Flux, Helm Operator, WKSctl, and Gimlet, Fleet based on GitOps compliance, Kubernetes nativeness, multi-cluster support, multi-repo and multi-tenancy, built-in UI, Kustomize integration, CI/CD approach, ease of use, and market/community presence. A similar symbolic rating scale (- - - to +++) is applied here to illustrate how each tool supports core GitOps practices.

TABLE 4.2: Comparison of GitOps Tools [63, 43]

Feature	ArgoCD	Flux	Helm Operator	WKSctl	Gimlet	Fleet
Core GitOps Compliance	+++	+++	+++	+++	+++	+++
Kubernetes Native	+++	+++	+++	+++	+++	+++
Multi-Cluster Support	+++	+++	+++	+++	0	+++
Multi-Repo & Multi-Tenancy	+++	+++	0	0	0	0
Built-In or Native UI	+++	—	—	—	++	0
Kustomize Integration	+++	+++	—	0	0	0
CI/CD Approach	0 (CD focus)	0 (CD focus)	0 (CD focus)	0	0	0
Ease of Use	+++	++	+	+	+++	+
Market Presence & Community	+++	+++	+	+	+	+

Explanation of Comparison Metrics (Table 4.2)

- **Core GitOps Compliance:** Indicates how closely each tool adheres to GitOps principles, such as storing declarative configurations in Git and using pull-based deployments.
- **Kubernetes Native:** Evaluate how tightly the tool integrates with Kubernetes APIs and resource management, reflecting ease of configuration and cluster administration.
- **Multi-Cluster Support:** Examines the tool's ability to manage multiple Kubernetes clusters from a single control plane. Higher ratings (+++) indicate seamless multi-cluster deployment and synchronization.
- **Multi-Repo & Multi-Tenancy:** Focuses on how well the tool handles multiple Git repositories and distinct user/team environments, including role-based access control and separate codebases.
- **Built-In UI:** Assesses whether the tool provides a native graphical interface for tasks like real-time monitoring, rollbacks, and operational management, beyond command-line tools.
- **Customize Integration:** Determines if the tool natively supports Customize for configuration overlays, facilitating environment-specific customizations without duplicating configuration files.
- **CI/CD Approach:** Reflects whether the tool is primarily CD-focused (pull-based deployments) Alternatively, it also includes CI capabilities. A "0" often indicates a CD-first approach may require separate CI solutions.
- **Ease of Use:** Rates the learning curve, clarity of documentation, and general user-friendliness of both the UI and CLI.
- **Market & Community:** Looks at adoption levels, frequency of releases, and user support channels (forums, GitHub, Slack, etc.). Tools with +++ typically have strong momentum and active user bases.

According to Table 4.2, ArgoCD consistently scores high (+++) in Kubernetes nativeness, multi-repo/multi-tenancy, and its built-in UI, making it a strong choice for organizations seeking an end-to-end GitOps solution. Flux also demonstrates excellent GitOps compliance (+++) and Kustomize integration (+++), though it lacks a native UI (—). Meanwhile, Helm Operator, WKSctl, and Gimlet have more limited capabilities (0 to +) in areas like multi-cluster support, often requiring additional tools or plugins. Overall, ArgoCD and Flux emerge as the most mature and feature-rich GitOps solutions.

Why ArgoCD?

ArgoCD was selected for this project due to its robust GitOps capabilities, including a native web interface, selective synchronization, and multi-cluster management. Its user-friendly UI allows real-time monitoring and simplified application management tasks, such as provisioning or rollbacks.

As highlighted in Table 4.2, ArgoCD's high (+++) ratings in multi-repo and multi-tenancy, Kubernetes nativeness, and ease of use underscore its suitability for diverse, production-scale environments. These features, combined with an active community and regular updates, make ArgoCD an excellent choice for organizations looking to implement or refine a GitOps workflow.

Chapter 5

Methodology and Implementation

This section details the methodology used to implement and evaluate traditional CI/CD and GitOps pipelines. To compare performance and operating efficiency, two pipelines were built using standard tools and best practices.

The methodology includes important steps for both traditional CI/CD pipelines using Jenkins and GitOps pipelines using ArgoCD.

5.1 Traditional CI/CD Pipeline Implementation

This section outlines the traditional CI/CD process for a Node.js application. The pipeline uses multiple tools, such as GitLab for version control, Jenkins for continuous integration and continous delivery, Docker for containerization, Kubernetes for orchestration, and Prometheus and Grafana for monitoring. The steps below detail the implementation approach for this pipeline.

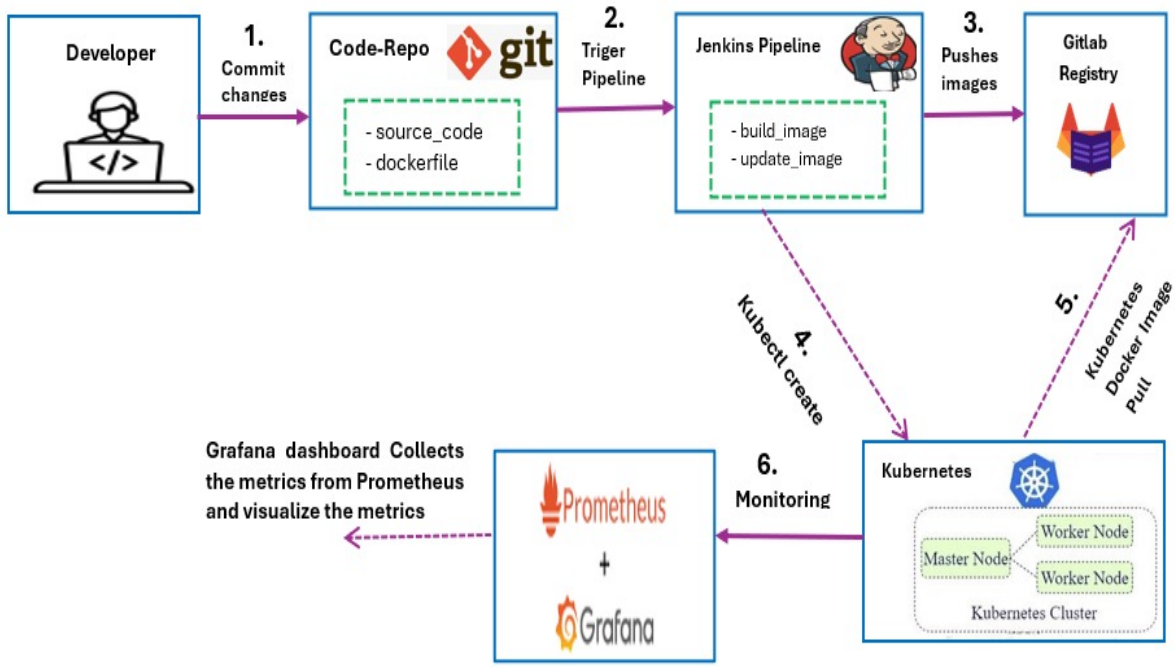

FIGURE 5.1: Traditional CI/CD

5.1.1 Repository and Code Management

To set up the Traditional CI/CD pipeline, first create a Git repository with GitLab for version control. GitLab was chosen because it seamlessly integrates with Jenkins for continuous integration jobs. The repository contained the Node.js application code, including source, configuration, and environment dependencies. GitLab uses a branching method, with the main branch representing production code and additional branches for feature development and bug fixes.

5.1.2 Provision Virtual Machines on OpenStack

To set up the infrastructure for hosting the CI/CD pipeline, OpenStack was used to provision virtual machines (VMs). OpenStack allows the management of virtualized resources such as compute, storage, and networking, providing a flexible and scalable platform for hosting the pipeline's components. VMs were created for running Jenkins, Docker, and Kubernetes. These VMs were configured within a private network, enabling seamless communication between Jenkins, Docker containers, and the Kubernetes cluster. The VMs provided the necessary resources to support the continuous integration and deployment processes, as well as the containerized application.

5.1.3 Docker for Containerization

Using Docker in this project supports a CI/CD process. Docker packages an application and its dependencies in a standardized container, allowing for seamless deployment across several environments. Integrating Docker with Jenkins allows for automated application development, testing, and deployment with minimal manual intervention.

```
# Use a lightweight Node.js 20 image for better performance
FROM node:16.0.0
#FROM node:20-alpine
#FROM node:22-slim
# Set the working directory in the container
WORKDIR /app

# Copy the package.json and package-lock.json files
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code to the container
COPY . .

# Expose the application port
EXPOSE 8080

# Define the command to run your application
CMD ["node", "app.js"]
```

FIGURE 5.2: Dockerfile

A Dockerfile was created to define the steps required to build the container image for the Node.js application.

5.1.4 Kubernetes Cluster Setup

Kubernetes is critical for automating application deployment and scalability in our continuous integration/delivery workflow. Jenkins handles continuous integration and deployment, automating the entire process from code commit to final shipment. Jenkins uses the appropriate Kubernetes manifests to manage the Node.js application's deployment directly on the cluster, improving efficiency, scalability.

Install Kubernetes Using Kubeadm: Kubernetes was installed using Kubeadm, which simplifies cluster setup. It first initializes the master node before adding the worker nodes, resulting in a scalable environment for application management.

Set Up the Kubernetes Deployment: To set up the Kubernetes deployment, configure a Deployment resource to handle the Node.js application with three replicas for high availability. This deployment retrieves the container image from the GitLab registry, ensuring that the application starts with the correct configuration.

```yaml
# manifests.yaml:

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: node-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: node-app
  template:
    metadata:
      labels:
        app: node-app
    spec:
      containers:
      - name: node-app
        image: container-registry.gitlab.cc-asp.fraunhofer.de/ipt_300/322edge/cicd_jenkins:latest
        ports:
        - containerPort: 3000
      imagePullSecrets:
      - name: gitlab-registry
```

FIGURE 5.3: Kubernetes Deployment File

Set Up the Kubernetes Service: A NodePort type service was defined to expose the application externally. This service maps port 3000 inside the container to 30007 on the node, allowing external access to the application.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: node-app
  ports:
    - port: 80
      targetPort: 3000
      nodePort: 30007
```

FIGURE 5.4: Kubernetes Service File

5.1.5 Configure Jenkins for CI/CD

After provisioning the infrastructure, Jenkins was set up as the continuous integration tool. Jenkins was integrated with GitLab, triggering builds anytime code was pushed to the repository.

```
pipeline {
    agent any
    environment {
        DOCKER_IMAGE_NAME = 'cicd_jenkins'
        GITLAB_REGISTRY_CREDENTIALS = 'gitlab-registry-token' // This  the credentials ID in Jenkins
        REGISTRY_URL = 'container-registry.gitlab.cc-asp.fraunhofer.de'
        PROJECT_NAME = 'ipt_300/322edge'
        IMAGE_TAG = 'latest'
        DEPLOYMENT_NAME = 'my-deployment'  // Name of Kubernetes Deployment
        K8S_NAMESPACE = 'default' // Namespace where the deployment is located
    }

    stages {
        stage('Cloning Code') {
            steps {
                echo "Cloning the repository..."
                script {
                    git(
                        url: "https://gitlab.cc-asp.fraunhofer.de/IPT_300/322edge.git",
                        branch: "cicd_jenkins",
                        credentialsId: GITLAB_REGISTRY_CREDENTIALS
                    )
                }
            }
        }
        stage('Prepare Build Environment') {
            steps {
                echo "Preparing the environment for building Docker image..."
                script {
                    sh """
                    echo "Setting up environment variables and prerequisites..."
                    """
                }
            }
        }
        stage('Build Docker Image') {
            steps {
                echo "Building the Docker image..."
                script {
                    sh """
                    docker build -t ${REGISTRY_URL}/${PROJECT_NAME}/${DOCKER_IMAGE_NAME}:${IMAGE_TAG} .
                    """
                }
            }
        }
        stage('Push Docker Image to GitLab registry') {
            steps {
                echo "Pushing the Docker image to the GitLab registry..."
                script {
                    withCredentials([usernamePassword(
                        credentialsId: GITLAB_REGISTRY_CREDENTIALS,
                        usernameVariable: 'USERNAME',
                        passwordVariable: 'PASSWORD'
                    )]) {
                        sh """
                        echo ${PASSWORD} | docker login -u ${USERNAME} --password-stdin ${REGISTRY_URL}
                        docker push ${REGISTRY_URL}/${PROJECT_NAME}/${DOCKER_IMAGE_NAME}:${IMAGE_TAG}
                        """
                    }
                }
            }
        }
    }
```

FIGURE 5.5: Jenkins CI Script

Checkout Stage: Check-out Stage: Jenkins pulls the most recent code from the GitLab repository using the checkout scm command. Jenkins uses the most up-to-date version of the code, taking into account repository updates and modifications.

Cloning Repository from GitLab: Clone repository from GitLab: During this stage, the pipeline copies the repository from GitLab. Jenkins authenticates and accesses the repository using stored credentials after specifying the URL and branch (cicd_jenkins). This step ensures Jenkins uses the correct code versions for the build process.

Build Docker Image: After the code is cloned, the pipeline moves forward to build a Docker image using the docker build command. The image is tagged with the

GitLab Container Registry URL and a version tag (latest). This step produces a containerized version of the application, which is essential for deployment.

Push Docker Image to GitLab Container Registry: After creating the Docker image, use the docker push command to upload it to the GitLab Container Registry. During this procedure, Jenkins authenticates with the registry using previously stored credentials, ensuring that the image is safely transferred and stored for future deployments.

The **Continuous Deployment (CD) pipeline** The CD pipeline automates deploying the application to a Kubernetes cluster through several steps:

```
    stage('Deploy to Kubernetes') {
        steps {
            echo "Deploying to Kubernetes server..."
            script {
                def remote = [
                    name: 'jenkins-cicd-master',
                    host: '10.32.199.50',
                    user: 'operation',
                    password: 'Master@12345678',
                    allowAnyHosts: true
                ]

                // Upload manifest
                sshPut remote: remote, from: 'manifest/manifests.yaml', into: '.'

                // Apply Kubernetes Manifest
                sshCommand remote: remote, command: "kubectl apply -f manifests.yaml"

                // Trigger rollout restart to use the new image
                sshCommand remote: remote, command: "kubectl rollout restart deployment ${DEPLOYMENT_NAME} -n ${K8S_NAMESPACE}"
            }
        }
    }

    stage('Pipeline Completion') {
        steps {
            echo "Pipeline execution completed successfully."
        }
    }
}
```

FIGURE 5.6: Jenkins CD Script

Upload Kubernetes Manifest to Remote Server:

At this phase, the Kubernetes manifest file (manifests.yaml), which contains the application's deployment configurations, is transferred to a remote server via the sshPut command. This guarantees that the required configuration is available on the server, allowing for a smooth deployment procedure.

Deploy Application to Kubernetes: The pipeline then deploys the application to Kubernetes by applying the manifest file with the kubectl apply command. This command creates or updates the application deployment on the Kubernetes cluster, ensuring that the latest Docker image is deployed according to the configuration in the manifest.

Verify Kubernetes Rollout: Finally, Jenkins verifies the deployment by checking the status of the Kubernetes rollout. The kubectl rollout status command is used to ensure the application is successfully deployed and running in the Kubernetes cluster. If the deployment fails, this step provides feedback to allow for troubleshooting.

Final Build a Jenkins CI/CD Pipeline: The Jenkins dashboard shown above provides a snapshot of a traditional CI/CD pipeline tailored for a Node.js application. Each column represents a different stage—such as cloning the repository, preparing the build environment, constructing the Docker image, pushing the image to the GitLab registry, and deploying to Kubernetes—while each row corresponds to a particular build run. This layout offers clear, stage-by-stage visibility into the pipeline's progress and average duration, enabling rapid identification of performance bottlenecks and facilitating informed decisions about future optimizations.

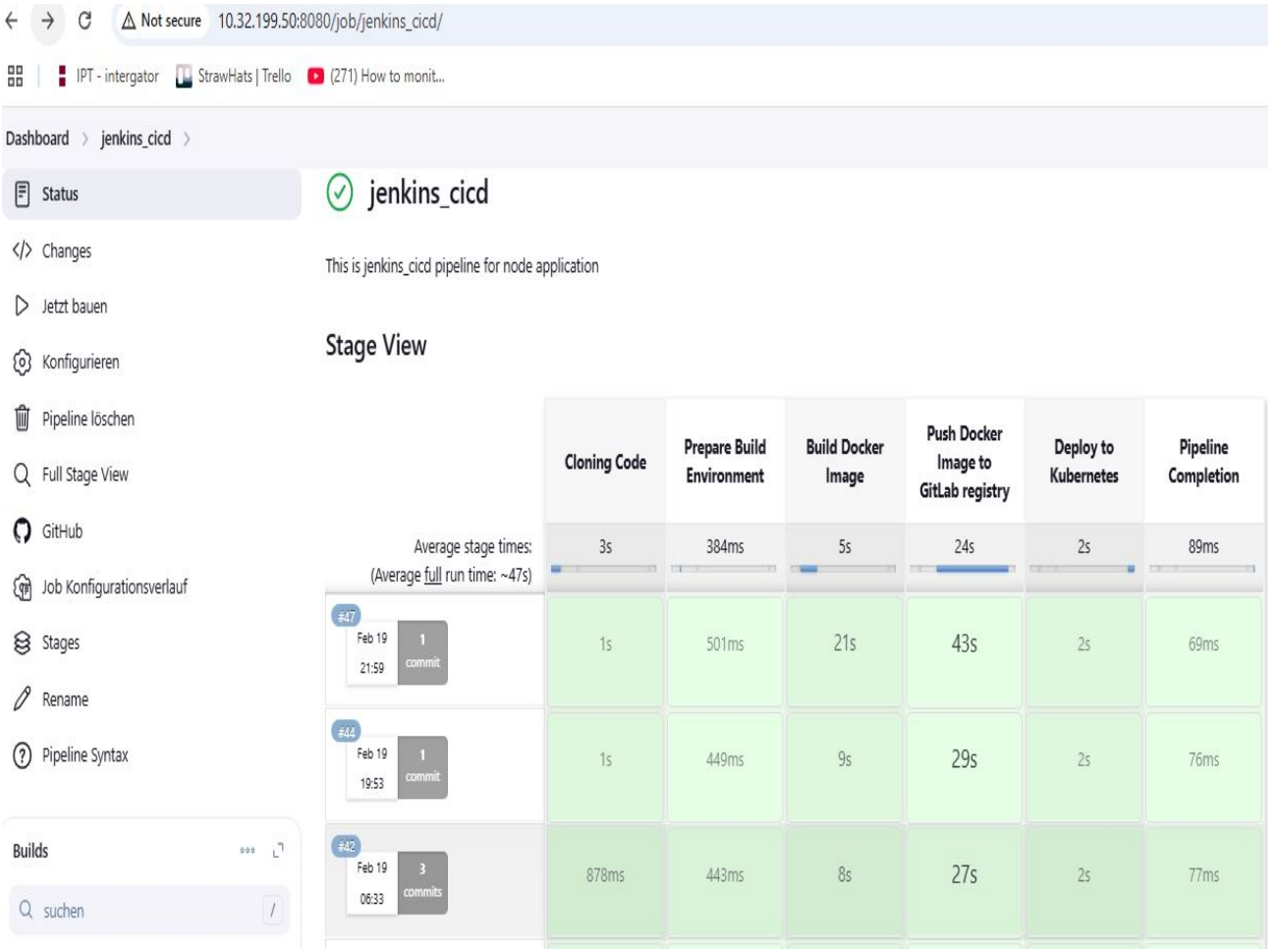

FIGURE 5.7: Pipeline build stages

The CI/CD workflow automates the build and deployment of node applications. In this setup, three different Node.js base images were tested to assess their overall impact on build and deployment times.

5.1.6 Monitoring and visualization

Prometheus and Grafana were integrated into the Kubernetes cluster to monitor both application performance and the underlying infrastructure metrics. This setup provides comprehensive insights into Jenkins CI/CD pipeline execution times and highlights potential bottlenecks within the system.

5.1.6a Prometheus Configuration

Prometheus was configured to scrape metrics from the Jenkins pipeline, various Kubernetes components (e.g., pods, nodes), and application endpoints. Specifically for Jenkins, a metrics endpoint was exposed to collect data on pipeline job duration. These metrics were stored in Prometheus's time-series database, enabling historical trend analysis and a clearer view of pipeline performance.

5.1.6b Grafana Dashboards

Grafana was used to create custom dashboards for visualizing the metrics collected by Prometheus. These dashboards provided real-time insights into application performance and facilitated the identification of bottlenecks. By analyzing these visualizations, it was possible to promptly detect performance issues and thereby enhance the efficiency of the CI/CD workflow.

FIGURE 5.8: Jenkins Pipeline Execution Time (node:16.0.0-324.04 MB)

The above diagram illustrates the performance of a Jenkins-driven CI/CD pipeline as monitored through Grafana. It displays the pipeline execution time and job performance metrics. At the peak observation, the Jenkins pipeline completed in approximately 69 seconds, representing the total execution time required for both the build and deployment stages. By integrating Jenkins with Grafana, real-time visibility into pipeline performance is achieved, enabling teams to quickly identify potential bottlenecks and optimize their CI/CD processes.

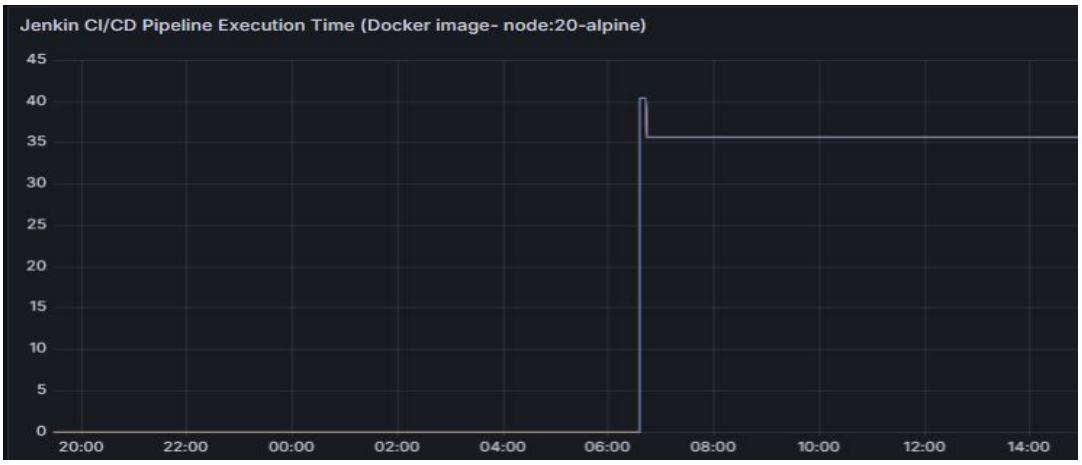

FIGURE 5.9: Jenkins Pipeline Execution Time (node:20-alpine-45.37 MB)

This diagram shows a consistent execution time plateau for the Jenkins CI/CD pipeline, again monitored via Grafana. The pipeline runs hover around 41 seconds, reflecting a uniform build and deployment duration throughout the observed period. Such stability suggests that the Alpine-based Node.js environment may offer a streamlined setup. As with the first diagram, this visibility into execution time aids in pinpointing inefficiencies and refining the pipeline configuration for faster, more reliable deployments.

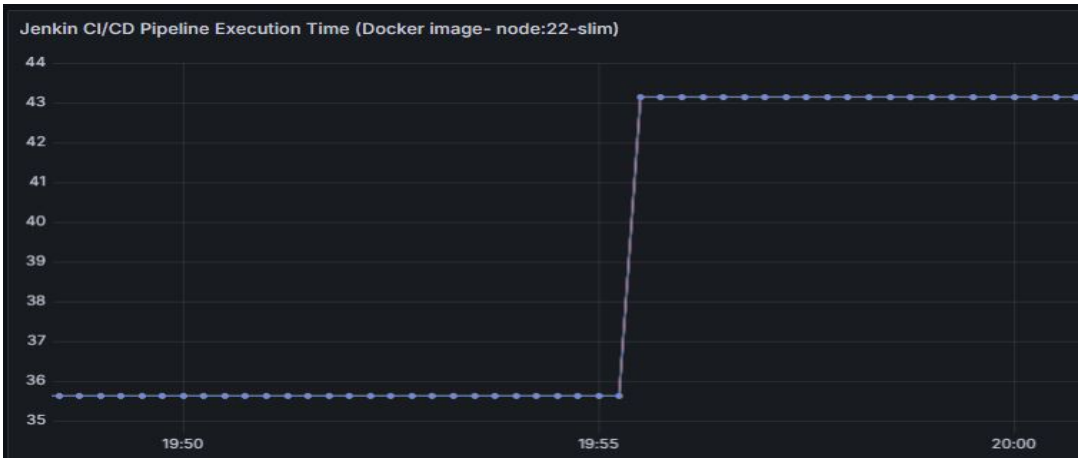

FIGURE 5.10: Jenkins Pipeline Execution Time (node:22-slim-74.14 MB)

In the final diagram, the Grafana dashboard indicates that the CI/CD pipeline completes in approximately 43 seconds. This measurement suggests a potential environmental, dependencies, or codebase alteration, emphasizing the critical need for continuous monitoring. By correlating pipeline execution time with the corresponding code commits or environment updates, developers can promptly identify and mitigate performance regressions.

Summary:

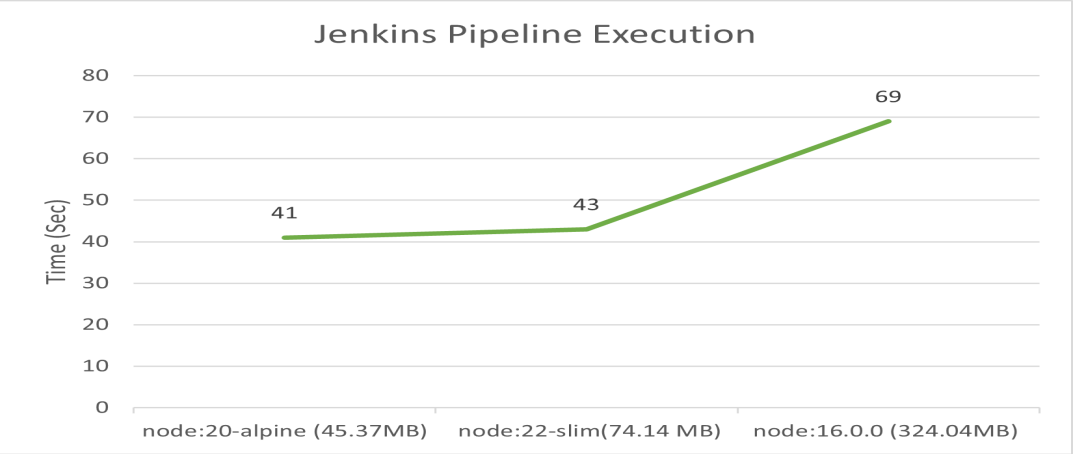

FIGURE 5.11: Execution Time of a Jenkins Pipeline for Various Node.js Versions

The graph illustrates the execution times of a Jenkins pipeline across three Node.js base images: Node.js 20-alpine (45 MB), Node.js 22-slim (74 MB), and Node.js 16.0.0 (324 MB). Node.js 16.0.0 demonstrates the longest execution time, suggesting that its larger image size may contribute to additional overhead during pipeline operations. By contrast, Node.js 20-alpine shows a substantial decrease in execution time, reflect- ing the performance benefits of a more compact Alpine-based environment. Although the execution time increases slightly with Node.js 22-slim, it remains lower than that of Node.js 16.0.0.

Overall, selecting an optimized, smaller container image—such as Node.js 20-alpine—is crucial to enhance the efficiency of the Jenkins pipeline. By minimizing image size, developers can reduce pull times, accelerate build and deployment steps, and ultimately achieve more streamlined CI/CD workflows.

5.2 GitOps Pipeline Implementation

Implemented the GitOps method to ensure a declarative and automated approach to managing deployments for the Node.js application. The process was carefully designed and executed to align with best practices, using tools like GitLab, ArgoCD, Kubernetes, Helm, Prometheus, and Grafana. Each implementation step is described below with appropriate figures to illustrate the workflow.

5.2.1 Repository Setup

A GitLab repository was set up as the single source of truth for the application code, Dockerfile, and the GitLab CI/CD pipeline configuration (.gitlab-ci.yml). This repository was structured into separate directories for application source code, containerization configurations, and deployment files. Developers committed all changes to this repository, which triggered highly efficient and reliable automated workflows.

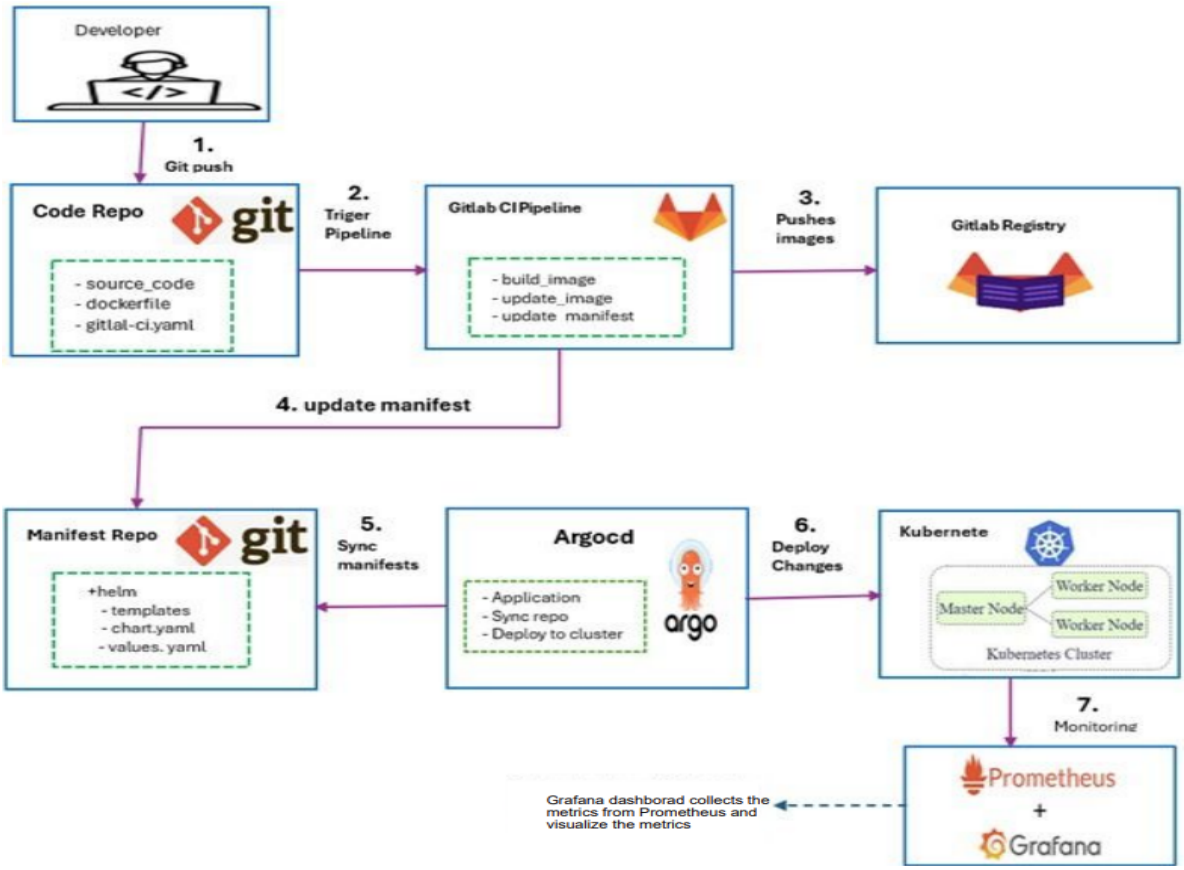

FIGURE 5.12: GitOps CI/CD

5.2.2 Provision virtual machines on OpenStack

Virtual machines (VMs) were provisioned using OpenStack to host the Kubernetes cluster and ArgoCD components. One VM was designated as the Kubernetes master node, while additional VMs acted as worker nodes.

5.2.3 Configure GitLab CI for CI/CD

The .gitlab-ci.yml file defines the CI/CD pipeline for the project. It includes stages like build, test, and deploy, with jobs specifying the scripts to run. GitLab Runners, the reliable backbone of our system, execute these jobs automatically whenever a commit is made, ensuring smooth and automated delivery.

```
variables:
  APP_NAME: research-project-thkoeln
  IMAGE_TAG: container-registry.gitlab.cc-asp.fraunhofer.de/ipt_300/322edge/$APP_NAME:$CI_COMMIT_SHORT_SHA

stages:
  - build
  - update_and_deploy

build_image:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
    - docker build -t "$IMAGE_TAG" -f "$CI_PROJECT_DIR/application/Dockerfile" "$CI_PROJECT_DIR/application"
    - docker push "$IMAGE_TAG"
```

Define Variables and Pipeline Stages: Begin by defining essential variables and pipeline stages in the .gitlab-ci.yml file. Variables like APP_NAME and IMAGE_TAG ensure consistent tagging and referencing of Docker images. The pipeline stages outline the sequential steps of the CI/CD process, such as building the image and updating the deployment manifests.

Build Docker Image and Push to Registry: In the build_image stage, log into the GitLab registry, build the Docker image using the defined tag, and push the image to the registry. This ensures that the latest version of the application is containerized and available for deployment.

```
update_and_deploy:
  stage: update_and_deploy
  image: ubuntu:22.04
  before_script:
    - apt-get update -y && apt-get install -y openssh-client git
    - mkdir -p ~/.ssh
    - echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
    - chmod 600 ~/.ssh/id_rsa
    - ssh-keyscan -H gitlab.cc-asp.fraunhofer.de >> ~/.ssh/known_hosts
    - eval $(ssh-agent -s)
    - ssh-add ~/.ssh/id_rsa
    - git config --global user.email "md.hazrat.ali@ipt.fraunhofer.de"
    - git config --global user.name "mdh02077"
    - git clone --branch research-project-thkoeln git@gitlab.cc-asp.fraunhofer.de:IPT_300/322edge.git
    - cd 322edge/manifests

  script:
    - sed -i "s/research-project-thkoeln:.*/research-project-thkoeln:${CI_COMMIT_SHORT_SHA}/g" test-login-app/values.yaml
    - git add test-login-app/values.yaml
    - git commit -m "Update image to ${CI_COMMIT_SHORT_SHA} [skip ci]" || echo "No changes to commit"
    - git push origin research-project-thkoeln
```

Update the Manifest File Using Helm: In the update_manifest stage, use Helm to update the Kubernetes manifest file with the new Docker image tag. This involves modifying the Helm values file to reference the latest image version and committing these changes to the GitLab repository. This ensures the deployment configuration is always updated with the latest application version, facilitating automated deployments.

5.2.4 Deploy to Kubernetes with ArgoCD

ArgoCD was used to implement continuous delivery in the GitOps workflow. ArgoCD was deployed to the Kubernetes cluster and configured to monitor the GitLab repository for changes in the Kubernetes manifests or Helm charts. The process involved

Connecting ArgoCD to the GitLab Repository:

A project in ArgoCD was linked to the GitLab repository, and the Git repository containing the manifests was linked to ArgoCD as an application. Sync policies were set to ensure automatic updates whenever changes were detected.

```
project: default
source:
  repoURL: https://gitlab.cc-asp.fraunhofer.de/IPT_300/322edge.git
  path: manifests/test-login-app
  targetRevision: research-project-thkoeln
  helm:
    valueFiles:
      - values.yaml
destination:
  server: https://kubernetes.default.svc
  namespace: default
syncPolicy:
  automated:
    prune: true
    selfHeal: true
  syncOptions:
    - CreateNamespace=true
```

Synchronizing the Deployment: ArgoCD applied the updated Kubernetes manifests to the cluster, ensuring that the actual state of the cluster matched the desired state defined in the repository.

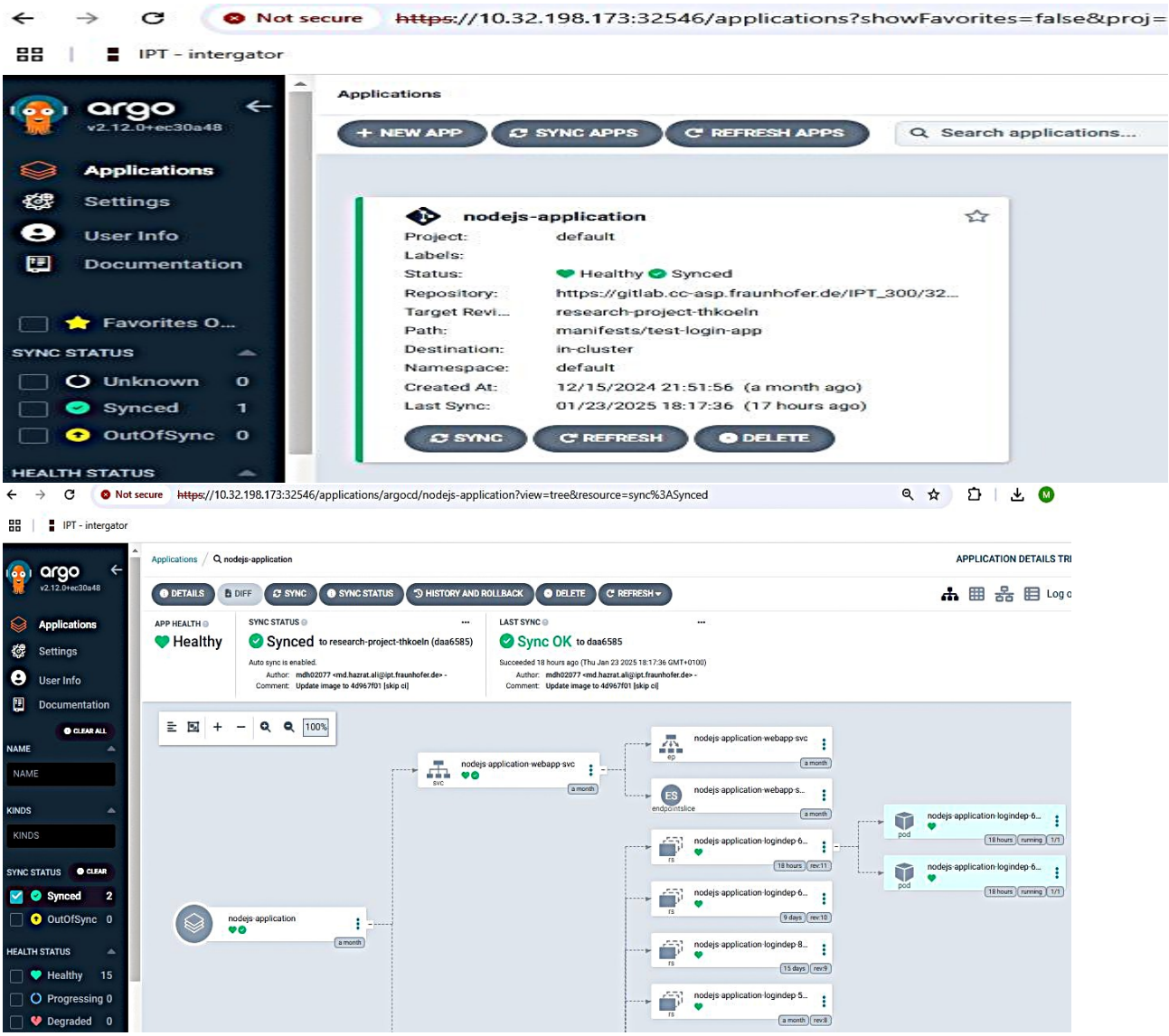

FIGURE 5.13: Argo CD Dashboard: Node.js Deployment to Kubernetes Cluster

The figure shows the Argo CD dashboard after the successful deployment of the

Node.js application to a Kubernetes cluster. It highlights key aspects of the deployment process and validates the effectiveness of the implemented workflow.

Application Overview: The application, named **nodejs-application**, is shown in a **healthy** and **synchronized** state. This indicates that all application components are functioning correctly and that the Kubernetes cluster fully aligns with the configuration stored in the Git repository. Synchronization ensures that updates made to the repository are automatically applied to the cluster.

Component Relationships: The hierarchical tree structure in the figure illustrates how the Kubernetes resources for the application are connected. It includes resources such as **services**, **endpoints**, **ReplicaSets**, and **pods**, all of which are running as expected. This structure provides a clear view of the application's architecture and dependencies.

Automation and Monitoring: Argo CD automates the deployment process by continuously monitoring the Git repository and ensuring the application remains configurable. Any changes in the repository are automatically applied to the cluster. The green health indicators in the figure confirm that all components are operating correctly.

Significance of the Setup: This deployment demonstrates how GitOps principles and Argo CD enable automated, reliable, and efficient application management. The figure validates the successful implementation of a GitOps workflow for deploying and managing the Node.js application in a Kubernetes environment, providing a robust and reliable system for our operations

5.2.5 Monitoring and visualization (Gitlab and ArgoCD)

The GitLab dashboard provides a real-time visualization of CI pipeline execution times, facilitating the prompt identification and resolution of performance anomalies. Integrating ArgoCD with Prometheus and Grafana within a Kubernetes environment enables comprehensive monitoring of application deployment durations and critical performance and infrastructure metrics. This combined approach gives a full, end-to-end picture of the CI/CD pipeline, making it easier to find bottlenecks and make improvements at the right time to ensure long-term pipeline reliability.

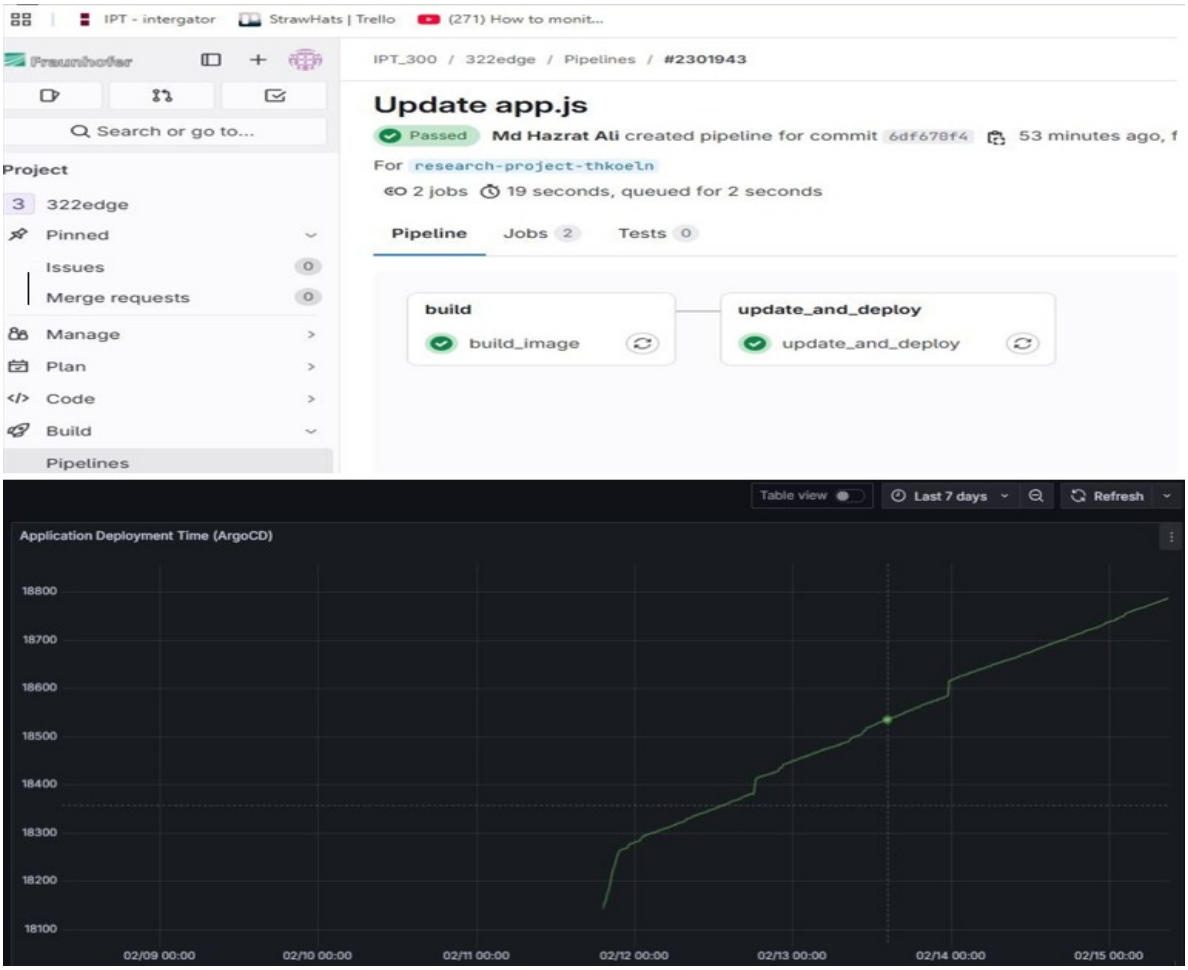

FIGURE 5.14: GitOps Pipeline Analysis(node:16.0.0 -324.04 MB)

Figure 5.14 presents a GitOps pipeline analysis, illustrating the integration of GitLab for Continuous Integration (CI) and ArgoCD for Continuous Deployment (CD). The pipeline deploys an application on node:16.0.0. The CI phase, including the build and *build_image* stages, is completed within 19 seconds. The *update_and_deploy* stage manages the image registry update and initiates the CD process. The total CI/CD pipeline execution time was approximately 37.7 seconds.

The ArgoCD application deployment, as depicted in the graph, shows a deployment duration reaching 18,700 ms. This deployment time reflects the automated synchronization of changes from the Git repository to the Kubernetes environment, facilitated by ArgoCD's declarative approach.

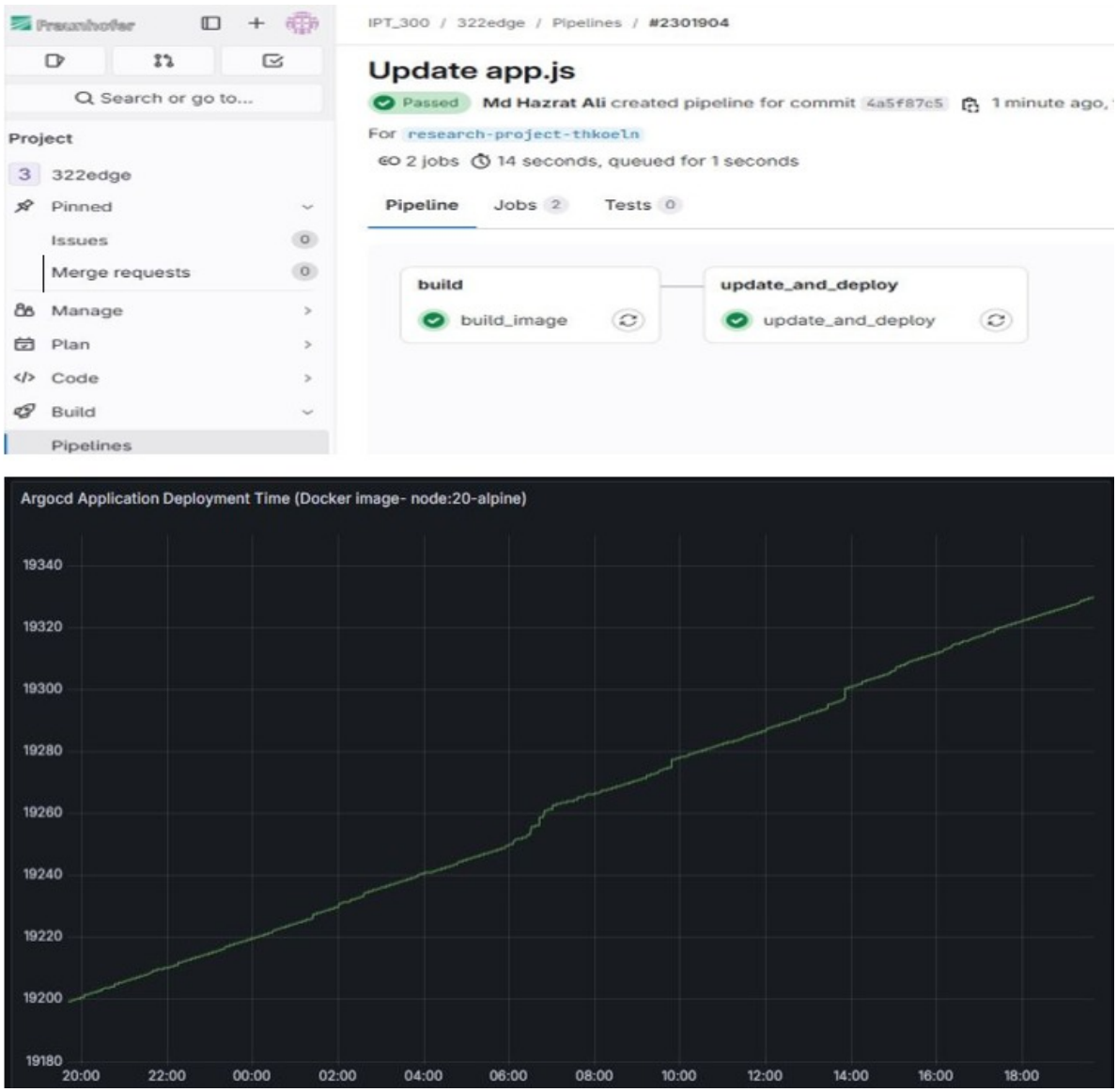

FIGURE 5.15: GitOps Pipeline Analysis (node:20-alpine-45.37 MB MB)

Figure 5.15 illustrates a GitOps pipeline wherein GitLab orchestrates Continuous Integration (CI) and ArgoCD facilitates Continuous Deployment (CD). The pipeline's total execution time is approximately 33.33 seconds, with the CI phase completing within 14 seconds and the deployment phase taking 19.33 seconds. The CI phase prepares the node:20-alpine Docker image, while the *update_and_deploy* stage manages the registry update and initiates the CD. The ArgoCD deployment, reaching 19660 ms, indicates a streamlined process, enabling continuous updates and consistent state management.

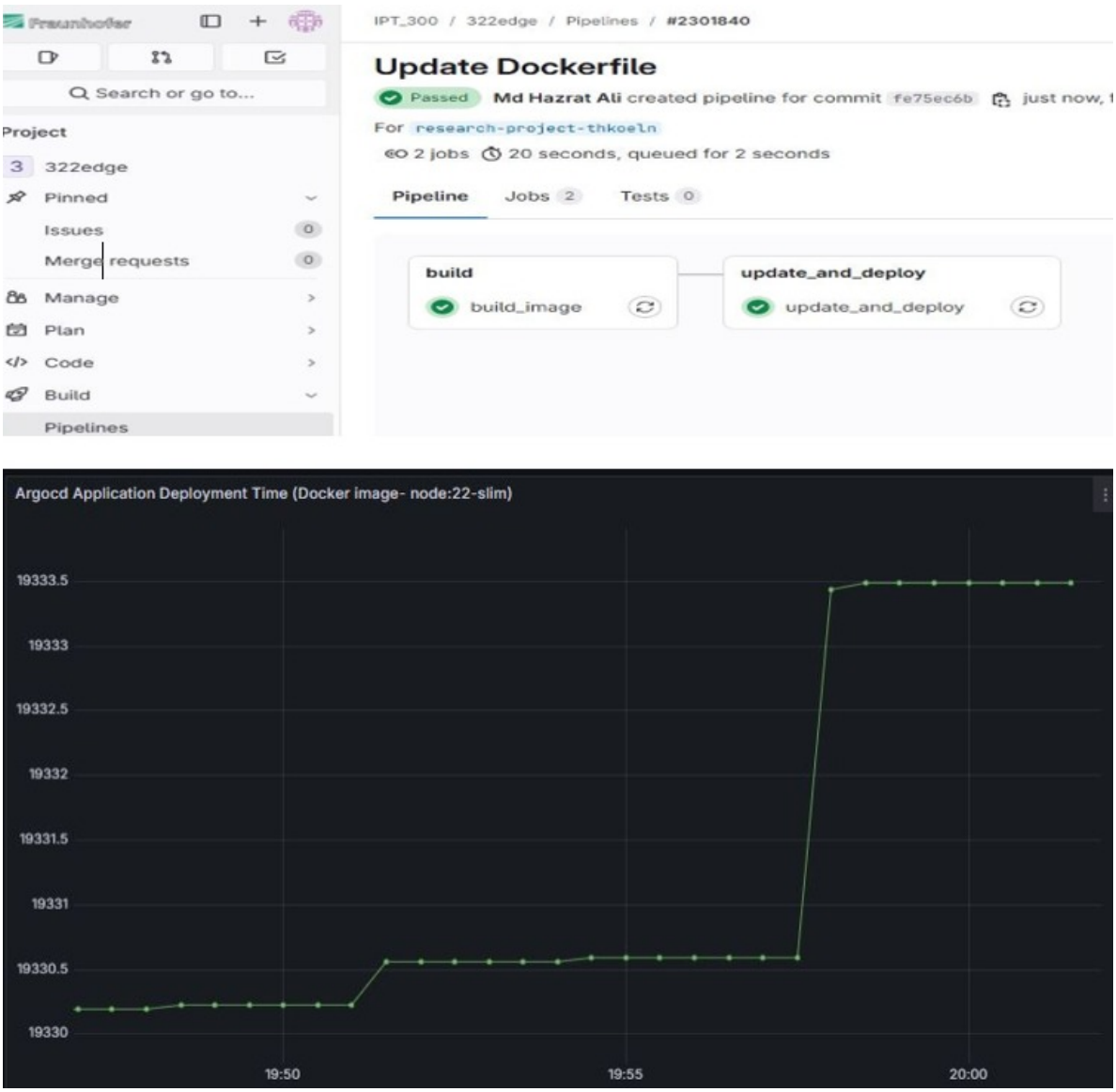

FIGURE 5.16: GitOps Pipeline analysis (node:22-slim -74.14 MB)

Figure 5.16 illustrates a GitOps pipeline, employing GitLab for Continuous Integration (CI) and ArgoCD for Continuous Deployment (CD), with deployment performance metrics visualized. The pipeline deploys a node:22-slim Docker image with a size of 74.14 MB. The CI phase, encompassing the build and *build_image* stage, as well as the image update and push to registry within the *update_and_deploy* stage, was completed in 20 seconds, as indicated in the pipeline summary. The CD phase, utilizing ArgoCD, is triggered following the completion of the *update_and_deploy* stage. The total CI/CD pipeline execution time is approximately 39.335 seconds.

The ArgoCD application deployment, as depicted in the graph, reaches 19333.5 ms. This rapid deployment, despite the larger image size, suggests the efficiency of ArgoCD's declarative approach. This methodology facilitates streamlined deployments within Kubernetes environments, enabling frequent updates. The observed performance highlights the potential for ArgoCD to enhance deployment speed and consistency."

Summary:

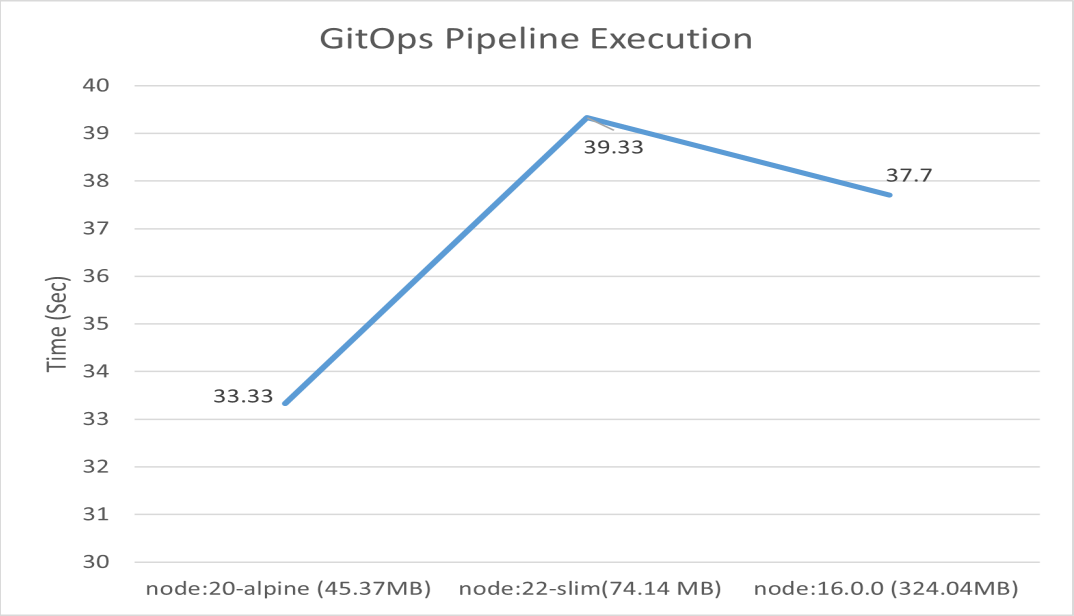

FIGURE 5.17: GitLab and ArgoCD CI/CD Pipeline analysis with different image size

The graph compares the execution times of a GitOps pipeline orchestrated through GitLab and ArgoCD across three Node.js base images: Node.js 16.0.0 (324 MB), Node.js 20-alpine (455 MB), and Node.js 22-slim (74 MB). Among these, Node.js 20-alpine achieves the shortest runtime, which can be attributed to its significantly smaller image size and lightweight Alpine-based environment. Conversely,Node.js 22-slim exhibits the longest runtime, suggesting additional overhead associated with its larger footprint compared to Node.js 20-alpine. Node.js 16.0.0 falls between these two extremes but still shows slower performance relative to the Alpine-based image. These findings underscore the impact of container image size on pipeline execution in GitOps workflows, indicating that selecting more optimized images particularly Node.js 20-alpine can substantially enhance overall efficiency.

Chapter 6

Comparative Analysis and Optimization Strategies

6.1 Comparative Analysis

In modern DevOps, CI/CD pipelines are indispensable for delivering software efficiently. This section offers a comparative analysis of a traditional Jenkins-driven pipeline and a GitOps-based approach using GitLab and ArgoCD, with performance metrics captured in real time via Grafana. In addition to evaluating how different Node.js base images—Node.js 16.0.0 (324.04 MB), Node.js 20-alpine (45.37 MB), and Node.js 22-slim (74.14 MB)—affect execution time, this analysis also considers the impact of container image size on resource utilization and overall pipeline responsiveness. By examining these configurations, the study highlights key trade-offs between Jenkins and GitOps, offering best practices for future CI/CD implementations. As shown in Figure 6.1, the observed execution times provide a direct basis for comparing the two methodologies.

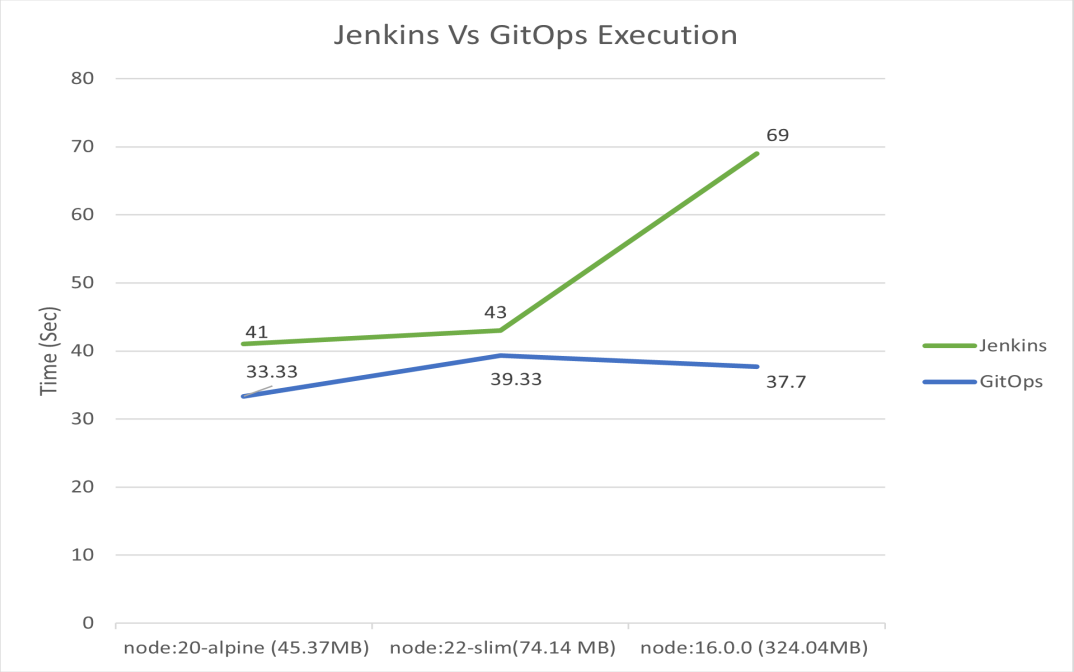

FIGURE 6.1: Comparison of Jenkins and GitOps Execution Times by Node.js Image Size

Figure 6.1 shows a comparison of the times it takes for Jenkins and GitOps (GitLab + ArgoCD) to run pipelines on various Node.js base images. Jenkins recorded the highest execution time when using Node.js 16.0.0, which also happens to be the largest image (324.04 MB). A significant decrease in execution time was observed with Node.js 20-alpine (45.37 MB), the smallest image in this comparison. There was a moderate increase in execution time with Node.js 22-slim (74.14 MB), indicating that while it is more optimized than Node.js 16.0.0, it does not match the streamlined performance offered by Node.js 20-alpine. Similarly, GitOps also experienced reduced runtime with Node.js 20-alpine compared to 16.0.0, followed by a slight rise with Node.js 22-slim. These results suggest that container image size plays a crucial role in pipeline efficiency: smaller images generally lead to quicker pull times, reduced overhead, and faster overall execution.

Furthermore, GitOps consistently outperformed Jenkins across all tested Node.js versions, indicating a reduction in execution times and operational overhead. Detailed measurements reveal that GitOps provides performance improvements ranging from approximately 8.53% (Node.js 22-slim) to 45.36% (Node.js 16.0.0), with an overall average of about 28% faster execution compared to Jenkins. This performance advantage stems from GitOps' declarative approach and automated synchronization mechanisms, ensuring that the infrastructure's desired and actual states remain aligned. By minimizing manual interventions and human error, GitOps streamlines the deployment process and leverages continuous monitoring and reconciliation capabilities to achieve accelerated deployments, enhanced reliability, and improved developer productivity. Overall, these findings suggest that GitOps is a more efficient and reliable choice than traditional Jenkins-based CI/CD pipelines, especially in environments where fast, consistent deployments are essential and where the choice of a smaller, more optimized container image such as Node.js 20-alpine can significantly impact performance.

6.2 Optimization Strategies

6.2.1 Recommended Optimizations in Traditional CI/CD Pipelines

The current Jenkins pipeline successfully builds and deploys a Node.js application to Kubernetes, fulfilling essential continuous integration and delivery (CI/CD) requirements. However, several strategies can be employed to enhance performance, scalability, and reliability further. These recommendations, such as optimizing build processes, implementing automated testing, and leveraging containerization, are based on established CI/CD best practices and address potential bottlenecks that may arise as project demands increase.

Here are key strategies, each supported by reputable sources: optimizing build processes, implementing automated testing, and leveraging containerization.

- **Implement Parallel Execution:** Currently, the pipeline executes stages sequentially, which can lead to increased build times as the number of tasks grows. By leveraging Jenkins' support for parallel execution, independent tasks (such as building Docker images and running tests) can be performed concurrently. This approach significantly reduces overall build durations, particularly in larger or more complex environments [66].
- **Utilize Caching Mechanisms:** Introducing caching strategies allows for the reuse of previously downloaded resources, thereby minimizing repetitive data

transfers and reducing build times [66]. Examples include Docker-layer caching and local dependency caching, both of which can expedite the build process and lower resource consumption.

- **Optimize Jenkins Agent Configuration:** The current pipeline employs agent any, suggesting reliance on a default or static agent infrastructure. However, Transitioning to containerized agents—such as Docker-based or Kubernetes-based agents—allows for dynamic scaling based on real-time workload demands. This efficient approach helps ensure optimal resource utilization and prevents bottlenecks that may arise when multiple builds require concurrent agent availability [66].
- **Streamline Testing Processes:** Incorporating a focused testing phase ensures critical test cases are executed before the application is deployed. Automated testing frameworks enhance consistency and reliability by detecting potential issues early, thus reducing the likelihood of failures in later stages of the pipeline [67].
- **Reduce Deployment Size:** Although the Docker image is built and pushed to a registry, there is no indication of measures taken to minimize its size. Employing lightweight base images (for instance, Alpine-based images) and removing unnecessary files can significantly reduce both build and deployment durations [67]. Smaller images not only lower transfer times between the registry and the Kubernetes cluster, but also reduce the overall resource consumption, thus improving the overall pipeline performance [67].
- **Implement Automated Rollbacks:** Integrating automated rollback strategies either through Kubernetes' native rollback functionality or a custom approach ensures minimal downtime and enhances reliability by swiftly restoring a known stable state if deployment issues occur [67].
- **Security Automation (DevSecOps):** Integrating Security Checks at Every Stage of the Pipeline is crucial for maintaining the robustness of CI/CD pipeline. Security automation, also known as DevSecOps, incorporates security procedures into the CI/CD pipeline to address vulnerabilities ahead of time. Tools like Snyk and Aqua Security can identify and remediate vulnerabilities during development, delivering strong security while preserving deployment speed. [65]

By applying these recommended optimizations, the existing Jenkins-based pipeline can evolve into a more robust, scalable, and efficient system. The result is faster software delivery, higher application availability, and a CI/CD workflow that better supports the demands of ongoing development and deployment activities.

6.2.2 Performance Optimization in GitOps Pipelines

- **Infrastructure as Code (IaC):** Infrastructure as Code (IaC): Manage infrastructure configurations, deployment manifests, and environment settings as code using tools like Terraform or Ansible to ensure consistency, repeatability, and scalability. [68]
- **Security and Compliance:** Enforce security policies, access controls, and compliance standards within Git repositories, CI/CD pipelines, and deployment

workflows using features like RBAC, secrets management, and automated audits to ensure secure, compliant deployments [68]

- **Enhancing ArgoCD Pipeline Efficiency:** Increasing the number of replicas for the ArgoCD server and application controller to distribute workload and improve responsiveness [69].
- **Leveraging Kubernetes-native Tools for Scalability:** Explain how Kubernetes tools like Horizontal Pod Autoscaler (HPA) and Custom Resource Definitions (CRDs) can enhance scalability and manage resources efficiently. [70, 71]
- **Implementing Caching Mechanisms:** Configure caching with appropriate expiration times to reduce repetitive data fetching and alleviate the load on the controller [69]
- **Streamlining Deployment Processes:** Automate and streamline deployment processes using GitOps principles, ensuring configuration changes are pushed to Git and the cluster state syncs accordingly. [72]
- **Utilizing Advanced Kubernetes Features:** Explore advanced Kubernetes features like operators and stateful sets to manage complex applications and ensure consistent deployment states. [70, 71]

Chapter 7

Conclusion

This research compared GitOps and traditional Jenkins-driven CI/CD pipelines to find the most efficient approach for Kubernetes environments. The results showed that GitOps, using GitLab and ArgoCD, was significantly faster, with shorter deployment times compared to traditional Jenkins pipelines. This highlights the advantage of automation and consistency in software delivery, especially for large, containerized applications.

For Fraunhofer IPT, adopting GitOps could lead to more reliable deployments. Using Prometheus and Grafana for real-time monitoring allows for quick identification of performance issues, resource use optimization, and high service availability maintenance. Additionally, having a centralized, declarative source of truth for configurations improves traceability and compliance, which is crucial for strong governance.

The study contributes to the theoretical understanding of cloud-native CI/CD pipelines and provides Fraunhofer IPT with practical recommendations for improving their DevOps practices. By implementing these strategies through full GitOps adoption or a hybrid approach, Fraunhofer IPT can better align its development pipelines with modern cloud-native principles. This alignment supports faster innovation, scalable infrastructure, and reduced human error, thereby enhancing the reliability and effectiveness of their digital transformation efforts.

In summary, the GitOps approach, with its automated and consistent methodology, has demonstrated significant improvements in efficiency and reliability compared to traditional Jenkins pipelines. This makes GitOps preferred for modern, scalable, and efficient software delivery in Kubernetes environments. For both CI/CD pipelines, the "node:20-alpine" base image was the most efficient, offering the shortest execution times. This image's smaller size and optimized dependencies likely contributed to building and deployment processes quicker, making it a practical choice for organizations aiming to enhance CI/CD pipeline performance.

References

[1] Tran Viet Thien, T. (2024, May 14). Analysing and designing CI/CD pipeline in an enterprise (Bachelor of Engineering thesis, Information Technology – Smart IoT Systems).

[2] Clement, Mateo. (2024). Comparative Analysis of GitOps vs. Traditional Infrastructure Management Approaches.

[3] Codefresh. "GitOps Workflow vs Traditional Workflow: What is the Difference?" [Online] Available: https://codefresh.io/learn/gitops/gitops-workflow-vs-traditional-workflow-what-is-the-difference/

[4] Ambassador Labs. "Reliable CI/CD Pipelines: Faster Software Releases," [Online] Available: https://www.getambassador.io/blog/reliable-ci-cd-pipelines-faster-software-releases

[5] BuildPiper. "The Art of CI/CD Optimization," [Online] Available: https://www.buildpiper.io/blogs/the-art-of-ci-cd-optimization/

[6] Red Hat Team. "What is CI/CD?," Red Hat, 2022, [Online] Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd

[7] Civo. "The Role of the CI/CD Pipeline in Cloud Computing," [Online]. Available: https://www.civo.com/blog/the-role-of-the-ci-cd-pipeline-in-cloud-computing

[8] Karuturi, A.B.S. (2021) An introduction to continuous integration, Qentelli. Available: https://qentelli.com/thought-leadership/insights/continuous-integration

[9] JetBrains Team. "TeamCity CI/CD Guide," JetBrains, [Online]. Available: https://www.jetbrains.com/teamcity/ci-cd-guide/

[10] "Thesis, Continuous Integration and Deployment," [Online] Available: https://www.theseus.fi/handle/10024/786622

[11] Roper, J. "CI/CD Pipeline: Everything You Need to Know with Examples," Spacelift, [Online]. Available: https://spacelift.io/blog/ci-cd-pipeline.

[12] "CI/CD on Automatic Performance Testing," ResearchGate, [Online]. Available: https://www.researchgate.net/publication/354399934/

[13] "Continuous Delivery – Jez Humble, David Farley," [Online]. Available: https://proweb.md/ftp/carti/Continuous-Delivery-Jez%20Humble-David-Farley.pdf

[14] Brigginshaw, D. "Guide to CI/CD Pipeline: Everything You Need to Know (2022)," Scriptworks, [Online]. Available: https://www.scriptworks.io/blog/ci-cd-pipeline/

[15] Anastasov, M. "CI/CD Pipeline: A Gentle Introduction," Semaphore, 2022, [Online]. Available: https://semaphoreci.com/blog/cicd-pipeline

[16] Katalon Team. "Benefits of Continuous Integration Delivery: CI/CD Benefits," Katalon, [Online]. Available: https://katalon.com/resources-center/blog/benefits-continuous-integration-delivery

[17] "Cloud-Native Continuous Integration," Journal of Computing and Information Technology, [Online]. Available: https://universe-publisher.com/index.php/jcit/article/view/3/3

[18] "Cloud-Native Development and Applications," Jaro Education, [Online]. Available: https://www.jaroeducation.com/blog/cloud-native-development-and-applications/

[19] "Event-Driven Delivery," sbstjn.com, [Online]. Available: https://sbstjn.com/blog/event-driven-delivery/.

[20] "GitLab CI Event Workflows," GitLab, [Online].
Available: https://about.gitlab.com/blog/2022/08/03/gitlab-ci-event-workflows/

[21] "What is GitOps?" GitLab, [Online]. Available: https://www.gitops.tech/

[22] "What is GitOps?" GitLab, Mar. 2023, [Online]
Available: https://about.gitlab.com/topics/gitops/

[23] A Beginner's Guide to GitOps and How It Works, GitLab, [Online]. Available: https://learn.gitlab.com/c/beginner-guide-gitops.

[24] "GitOps: The Evolution of DevOps," IEEE Xplore, [Online]. Available: https://ieeexplore.ieee.org/document/9565152

[25] "Weaveworks GitOps AWS Workshop," [Online]. Available: https://weaveworks-gitops.awsworkshop.io/.

[26] S. Chacon and B. Straub, Pro Git, 2nd ed. Apress, 2014. [Online]. Available: https://git-scm.com/book/en/v2

[27] "GitHub," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/GitHub

[28] GitLab Handbook. GitLab Documentation, [Online]. Available: https://docs.gitlab.com

[29] "Building an Open Source Company, [Online]. Available: https://about.gitlab.com/blog/2016/07/14/building-an-open-source-company-interview-with-gitlabs-ceo/

[30] Wikipedia Contributors, "Docker (software)," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software)

[31] "Architecture of Docker," GeeksforGeeks, [Online]
Available: https://www.geeksforgeeks.org/architecture-of-docker/.

[32] "Kubernetes Architecture," Kubernetes Documentation, [Online]
Available: https://kubernetes.io/docs/concepts/architecture/

[33] "Kubernetes Architecture," VMware, [Online]
Available: https://www.vmware.com/topics/kubernetes-architecture

[34] "Introduction to Kubernetes," PSJ Codes, [Online]. Available: https://psj.codes/introduction-to-kubernetes#heading-what-is-kubernetes

[35] "Helm Documentation," [Online]. Available: https://helm.sh/docs/

[36] What is a Helm Chart? Tutorial for Kubernetes Beginners," freeCodeCamp, [Online]. Available: https://www.freecodecamp.org/news/what-is-a-helm-chart-tutorial-for-kubernetes-beginners/.

[37] "Jenkins Official Website," [Online]. Available: https://www.jenkins.io/.

[38] "Automating CI/CD with Jenkins," Jenkins Documentation, [Online]. Available: https://jenkins.io/doc/book/

[39] "ArgoCD Official Documentation," [Online]. Available: https://argo-cd.readthedocs.io/en/stable/

[40] "GitOps Continuous Delivery with ArgoCD," Red Hat, [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-argocd.

[41] "Prometheus Documentation," [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[42] "Grafana Documentation," [Online]. Available: https://grafana.com/docs/grafana/latest/

[43] Push-based Deployments, " [Online]. Available: https://www.gitops.tech/

[44] "Official Bamboo Page," Atlassian, [Online]
Available: https://www.atlassian.com/software/bamboo/

[45] P. Polkhovskiy, Thesis, Tampere University, [Online]
Available: https://trepo.tuni.fi/bitstream/handle/123456789/24043/polkhovskiy.pdf

[46] "Publication on Continuous Integration and Deployment," ResearchGate, [Online]. Available: https://www.researchgate.net/publication/386183086

[47] "CircleCI Continuous Integration for Web Apps," StackShare, [Online]. Available: http://stackshare.io/circleci

[48] "Continuous Integration and Deployment," CircleCI, [Online]. Available: https://circleci.com/.

[49] "Codeship," G2 Crowd, [Online] Available:
https://www.g2crowd.com/products/codeship/reviews

[50] "Codeship - Continuous Integration and Delivery Made Simple," StackShare, [Online]. Available: http://stackshare.io/codeship

[51] "TeamCity is an Ultimate Continuous Integration Tool for Professionals," StackShare, [Online]. Available: http://stackshare.io/teamcity

[52] "Travis CI - A Hosted Continuous Integration Service for Open Source and Private Projects," StackShare, [Online]. Available: http://stackshare.io/travis-ci

[53] "Travis CI Documentation," [Online]. Available: https://docs.travis-ci.com/.

[54] "Jenkins - An Extendable Open Source Continuous Integration Server," [Online].
Available: http://stackshare.io/jenkins

[55] "Jenkins - An Extendable Open Source Automation Server," Jenkins CI, [Online].
Available: https://jenkins-ci.org/

[56] A. Bayer, "Jenkins Blog," [Online]. Available: https://www.jenkins.io/blog/authors/abayer/.

[57] "Flux Documentation," FluxCD, [Online]. Available: https://fluxcd.io/flux/.

[58] D. Szakallas, "Comparison: Flux vs Argo CD," Earthly, [Online]. Available: https://earthly.dev/blog/flux-vs-argo-cd/.

[59] M. Klinka, GitOps Principles for Deployment and Management of the AFoLab Platform, Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2024, [Online]. Available: https://is.muni.cz/th/kcqfv/.

[60] "ArgoCD Documentation," [Online]. Available: https://argo-cd.readthedocs.io/en/stable/.

[61] R. Portela and Á. Sándor, "Comparing GitOps Tools," 2020, [Online]. Available: https://blog.container-solutions.com/fluxcd-argocd-jenkins-x-gitops-tools.

[62] "Rancher Fleet - GitOps at Scale," [Online]. Available: https://fleet.rancher.io/.

[63] "Webthesis," Politecnico di Torino, [Online]. Available: https://webthesis.biblio.polito.it/18142/.

[64] "CI/CD Pipeline with Jenkins," AccelQ, [Online]. Available: https://www.accelq.com/blog/ci-cd-pipeline-with-jenkins/.

[65] "Optimizing CICD in Healthcare: Techniques for Streamlined," ResearchGate [Online] Available: https://www.researchgate.net/publication/386497419

[66] "Strategies for Improving Jenkins Pipeline Performance: Best Practices and Implementation," dev.to, [Online]. Available: https://dev.to/bcherlapally/strategies-for-improving-jenkins-pipeline-performance-best-practices-and-implementation-2309.

[67] "Optimize Your CI/CD Pipeline for Faster Deployments," Microtica, [Online]. Available: https://www.microtica.com/blog/optimize-your-ci-cd-pipeline-for-faster-deployments/.

[68] "GitOps Pipeline Optimization," Prodshell, [Online]. Available: https://prodshell.com/pages/gitops-pipeline-optimization.html.

[69] "Optimizing ArgoCD Performance," Hossted, [Online]. Available: https://hossted.com/knowledge-base/case-studies/devops/developer-tools/optimizing-argo-cd-performance/

[70] "Paper Details," IRE Journals, [Online]. Available: https://www.irejournals.com/paper-details/1705127?form=MG0AV3

[71] "Containerization and Kubernetes: Scalable and Efficient Cloud-Native Applications," IJISRT, [Online]. Available: https://ijisrt.com/containerization-and-kubernetes-scalable-and-efficient-cloudnative-applications?

[72] "Introducing ArgoCD: A GitOps Approach to Continuous Deployment," dev.to, [Online]. Available: https://dev.to/sre_panchanan/introducing-argocd-a-gitops-approach-to-continuous-deployment-p8e.