POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

MASTER's Degree Thesis

Thesis TITLE Comparative Analysis & Implementation of Modular CI/CD for Industrial IoT Solutions: Tool Evaluation, Selection, and Pipeline Deployment on Microservices Platforms

Supervisors

Prof. Marco TORCHIANO

Stefano IACONELLI

Candidate

Maaham BANU

MARCH 2026

Comparative Analysis & Implementation of Modular CI/CD for Industrial IoT Solutions: Tool Evaluation, Selection, and Pipeline Deployment on Microservices Platforms

Maaham Banu

Abstract

Continuous Integration and Continuous Delivery/Deployment (CI/CD) constitute a fundamental paradigm in modern cloud-native and DevOps-driven software engineering. These practices automate the code integration, testing, validation, and deployment processes, thereby improving development efficiency and reducing operational risk. As contemporary distributed software ecosystems increasingly adopt microservice-based, containerized, and cloud architectures, CI/CD pipelines have become essential to ensure rapid iteration, reproducibility, scalability, and system reliability. Consequently, the performance, scalability, and architectural design of CI/CD platforms significantly influence the development speed and overall system stability. Although CI/CD platforms are widely adopted in industry, limited research has systematically evaluated their performance within complex distributed environments such as microservice-based Industrial IoT (IIoT) systems. This study presents a structured comparative analysis of two commonly used CI/CD platforms, Jenkins and GitLab CI/CD. The evaluation is based on quantitative performance indicators, including build time, resource utilization, parallelization efficiency, scalability, tool sability and failure or error rates. In addition to performance benchmarking, the study examines architectural flexibility, modularity, and operational complexity to provide practical guidance for tool selection in industrial microservice platforms. Beyond comparative evaluation, this research designs and implements a modular end-to-end CI/CD framework tailored for Industrial IoT solutions. The proposed pipeline integrates automated unit testing, infrastructure-as-code (IaC) validation and automated environment provisioning, static code quality analysis using SonarQube, and AI-assisted stakeholder report generation through Amazon Bedrock. The pipeline concludes with automated email-based reporting, enhancing transparency and governance within the DevOps workflow. This implementation serves both as an experimental benchmark environment and as a proof-of-concept for modular CI/CD architectures in distributed industrial systems. The findings contribute empirical insights into performance trade-offs and architectural considerations when deploying CI/CD pipelines in industrial cloud-native environments.

Keywords: CI/CD; DevOps; Industrial IoT; Microservices Architecture; Jenkins; GitLab CI; Modular Pipelines; Cloud-Native Systems; Infrastructure as Code; Performance Evaluation; AWS; Sonar Qube; Serverless Computing; AI-Driven Automation; Automated Reporting.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Prof. Marco Torchiano for his guidance and support throughout the development of this thesis. I am also deeply thankful to Stefano Iaconelli, my manager, for his support during the implementation and design phases of this work and for providing valuable professional insight throughout the process. I would like to thank my dear friends Ikbal, Maryam, Shafeo, Sevara, Laura, Nazanin, and Nazira for their encouragement and support during this journey. My deepest gratitude goes to my mother, Anisa, my aunt Saira, who has been my constant support, and to all my uncles and aunts who were always there for me.

Finally, I dedicate this work to my late father, whose dreams and belief in me continue to inspire me every day.

Table of Contents

1 Introduction **1**

1.1 Problem Statement 1

1.1.1 Practical Problem Context 2

1.2 Need for Study 3

1.3 Objectives 3

1.3.1 Research Questions 4

1.4 Methodology 4

1.5 Limitation of Study 4

1.6 Thesis Organization 5

2 Literature Review **6**

2.0.1 Evolution of CI/CD and Industry Adoption of DevOps Methdologies 7

2.0.1.1 Jenkins 7

2.0.1.2 Gitlab CI 7

2.0.2 Challenges in CI/CD Implementation 7

2.1 CI/CD in Distributed and Cloud-Native Systems 8

2.2 Performance Metrics in CI/CD 9

2.2.0.1 Execution Time 10

2.2.0.2 Resource Utilization 10

2.2.0.3 CPU and Memory Usage 10

2.2.0.4 Error Rates 11

2.2.0.5 Tool Usability 11

2.2.0.6 Scalability 11

2.3 Selection of Tools 12

2.4 Comparison of Selected Tools 12

2.4.0.1 Jenkins: A Legacy CI/CD Tool 12

2.4.0.2 Gitlab CI: A centralised DevOps Ecosystem 13

2.5 Comparative Analysis and Challenges 14

2.6 Research Gap in Industrial IoT CI/CD and Future Implications . . . 15

3 Pipeline Design and Setup **16**

3.1 Infrastructure Architecture Review 16

3.2 System Architecture Components 17

3.2.0.1 Field Infrastructure 18
3.2.0.2 MQTT Broker 18
3.2.0.3 Rule Engine 18
3.2.0.4 Message Queue (SQS) 18
3.2.0.5 Serverless Processing Functions 18
3.2.0.6 Database Layer 18
3.2.0.7 WebSocket Communication Layer 19
3.2.0.8 API Gateway 19
3.2.0.9 Frontend Application and Content Delivery 19
3.2.0.10 Developer Access and Management 19
3.3 Pipeline Design and Architecture 19
3.3.0.1 CI/CD Pipeline Workflow 19
3.3.0.2 Sonarqube Integration 20
3.3.0.3 Unit and E2E testing 20
3.3.0.4 Serverless Reporting System 21
3.4 Pipeline Setup 25
3.4.0.1 Gitlab Pipline Setup 25
3.4.0.2 Jenkins Pipeline Setup 28
3.5 Differences Between GitLab CI and Jenkins Pipeline Setups 30
3.5.1 Pipeline Definition 30
3.5.2 Infrastructure Management 31
3.5.3 Parallel Execution Model 31
3.5.4 Authentication and Environment Preparation 32
3.5.5 Deployment Workflow 32
3.5.6 Reporting and Artifact Handling 32

4 Comparative Analysis and Results **34**
4.0.1 Build Time 34
4.0.1.1 Jenkins 34
4.0.1.2 Gitlab CI 35
4.0.2 Resource Utilization 36
4.0.2.1 Jenkins 36
4.0.2.2 Gitlab CI 36
4.1 Error Rate 36
4.1.0.1 Jenkins 37
4.1.0.2 Gitlab CI 37
4.2 Usability 38
4.3 Job Execution and Parallelization 38
4.3.0.1 Jenkins 38
4.3.0.2 Gitlab CI 38
4.4 Scalability 39
4.4.0.1 Jenkins 39
4.4.0.2 Gitlab CI 39

4.5 Cost Analysis of CI/CD Tools 39
4.5.1 Jenkins Cost Analysis 39
4.5.2 Gitlab Cost Analysis 40
4.6 Evaluation of the Serverless AI Reporting System 40
4.6.0.1 Initial Prompt Configuration 41
4.6.0.2 Simplified Prompt Configuration 41
4.6.0.3 Final Prompt Configuration 42
4.6.0.4 Observations 44
4.7 Cost Analysis of the Serverless Reporting System 44

5 Conclusion **45**

Bibliography **47**

Dedications **49**

List of Figures

1.1 Methodological Workflow for CI/CD Pipeline Evaluation 5

3.1 Experimental Infrastructure for CI/CD Evaluation 17
3.2 AWS based Distributed Architecture 21
3.3 Serverless Reporting System 23
3.4 Folder Structure in the S3 bucket 23
3.5 Test Results of Each Stack 24
3.6 .XML file of VPC Stack 24
3.7 Serverless AI Reporting Stack Deployed using Terraform 24
3.8 Gitlab Pipeline Execution Structure 25
3.9 Gitlab Pipeline: Stages and Jobs 28
3.10 Jenkins Pipeline: Stages and Jobs 30
3.11 Gitlab Pipeline: Stages and Jobs 31

4.1 Initial Prompt Configuration Results 41
4.2 Simplified Prompt Configuration Results 42
4.3 Final Prompt Configuration Results 43

List of Tables

3.1 Comparison of GitLab CI and Jenkins Pipeline Implementations . . 33

4.1 Comparative Analysis of Jenkins and GitLab CI 35
4.2 Compute Resources Used for CI/CD Pipeline Experiments 36
4.3 Comparative Analysis of Error Rates of Jenkins and GitLab CI . . . 37
4.4 Cost Comparison of Jenkins and GitLab CI Pipeline Execution . . . 40
4.5 Estimated cost per AI-generated CI report 44

Chapter 1

Introduction

1.1 Problem Statement

Global data generation has increased at a rate never seen before due to the quick development of digital technology, networked devices, and data-driven applications. According to recent estimates, the amount of data generated globally in 2024 was estimated to be over 149 zettabytes [1]. By 2025, this amount is expected to rise to about 181 zettabytes per year. The growing use of cloud computing, Internet of Things (IoT) devices, and real-time data processing systems is a major factor in this exponential rise [1] [2].

IIoT platforms have to handle communication between a variety of physical devices and software services that operate in dispersed environments in addition to processing massive amounts of data. Coordination between hardware systems, data pipelines, backend services, and cloud infrastructure is frequently necessary for the integration of these disparate components. As the number of devices and services grows, maintaining stable deployments, ensuring component compatibility, and managing system updates become more challenging. Manual management of such systems increases the likelihood of operational risks, integration failures, and configuration errors.

Large-scale data processing and the integration of various devices require a high degree of automation and minimal human involvement to ensure dependability and operational stability. Automated processes are particularly important in industrial settings where system failures or inconsistencies could disrupt critical operations. However, many industrial systems still rely on traditional software delivery workflows, which have relatively slow deployment cycles and manual updates. When compared to traditional web applications, IIoT deployments often require additional validation steps, infrastructure dependencies, and coordination across multiple system layers, which can significantly slow down development and release cycles.

Managing the complexity of modern IIoT architectures is often too much for traditional development and deployment workflows. These workflows can cause problems like late updates, environments that aren't always the same, and trouble coordinating changes across services that are spread out. These inefficiencies make

operations more complicated and slow down development as IIoT systems grow and change.

Agile and DevOps practices can help with these problems. The agile methodology emphasizes incremental modifications and progressive development. This means that systems can change over time with small, controlled updates instead of big, rare releases. So, for complex distributed systems like Industrial IoT, incremental development methods can help cross-functional teams make changes slowly while keeping the system stable and lowering the risk that comes with big deployments.

DevOps practices make this approach even better by getting people from different areas, like development, operations, and infrastructure, to work together. In most traditional software development environments, different people are in charge of different parts of the project. IIoT platforms, on the other hand, need software engineers, infrastructure specialists, data engineers, and industrial system operators to work together. You need strong automation tools to make sure that everything is the same in all three environments: development, testing, and production. This will keep these teams up to date.

Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines are very important for this kind of automation to work. CI/CD pipelines automate the steps of integration, testing, and deployment, which makes it more likely that system updates will work requiring little human intervention. In Industrial IoT settings, CI/CD pipelines help make sure that infrastructure is always set up the same way, that system components are automatically checked, and that microservices that handle streams of industrial data are always deployed correctly. These features help development teams manage complicated distributed systems while keeping operations running smoothly.

Even though CI/CD implementation in DevOps practices are becoming more important in modern software engineering, there has not been much research on how CI/CD platforms work in Industrial IoT settings with distributed microservices architectures, diverse device ecosystems, and high data throughput. It is very important for engineers to pick the right CI/CD tools and make automation pipelines that can grow with the system as more and more companies use cloud-native and microservice-based architectures for IIoT systems. This study meets this requirement by doing a comparative analysis and implementing modular CI/CD pipelines that are specifically made for Industrial IoT solutions.

1.1.1 Practical Problem Context

This study is predicated on a practical implementation scenario encountered during the development of an Industrial IoT platform comprising multiple microservices for data ingestion and processing, in addition to the aforementioned significant research challenges. There had to be dependable ways to test, validate the infrastructure, deploy, and send system updates in order to manage the development lifecycle of these services.

There were a lot of steps in the development and deployment process that had to be done by hand before an automated solution was put in place. For things like running tests, checking infrastructure settings, and reporting deployment results, different workflows were used. This made it more likely that things would go wrong and made the release cycles take longer.

This study also addresses how to set up a modular end-to-end CI/CD pipeline that automates important parts of the development life-cycle in order to deal with these problems. The CI/CD pipelines utilizes AI to combine automated testing, code quality analysis, infrastructure validation, and automated reporting. Making deployments more reliable and easier for cross-functional teams to understand, debug or improve. This example shows how CI/CD automation can help build scalable and reliable Industrial IoT development environments in real life, where there are multiple stakeholders involved.

1.2 Need for Study

As software systems are becoming more complex, the way software is made, tested, and deployed has changed a lot. More and more businesses are using DevOps and cloud-native architectures to speed up delivery times, add new features all the time, and make systems more stable. In this case, Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines are now a key part of how modern software is made.

Continuous Delivery/Deployment (CI/CD) pipelines have become a standard in the software development industry. However, setting up the CI/CD workflow can be challenging when multiple diverse components are involved such as edge-devices integrated with cloud platform which involves multiple infrastructure dependencies. Therefore, ensuring the correct infrastructure setup, understanding the dependencies before the implementation of a CI/CD pipeline is highly crucial. Taking the practical consideration into accounts it's very important for the right tool selection.

1.3 Objectives

The principal objective of this study is to ascertain how modular CI/CD pipelines can enhance automation, reliability, and operational efficiency in Industrial Internet of Things (IIoT) systems that utilize microservice-based architectures.

The thesis aims to evaluate different tools in the CI/CD landscape by comparing build times, resource utilization, scalability, tool usability, error rates, and overall pipeline efficiency of Jenkins and Gitlab CI. Moreover, key aspects of tool integration and tool usability will also be studied and compared. In the comparitive analysis the thesis aims to provide thorough analysis and key insights on strengths, tradeoffs, and limitations. Therefore, the main objective of the thesis is to perform a comparative analysis using an existing infrastructure. The thesis also aims to provide the key components involved in the design of the CI/CD pipeline for a more comprehensive

analysis. Therefore, enabling students and developers understanding the modern CI/CD workflows in the DevOps landscape particularly in the IIoT sector.

1.3.1 Research Questions

This thesis aims to investigate the following research questions:

- **Research Q1:** How well do Jenkins and GitLab CI work with CI/CD workflows in Industrial IoT settings with diverse components?
- **Research Q2:** How do CI/CD platforms compare to each other in terms of how long it takes to run a pipeline, how much resources it uses, and how well the pipeline works overall?
- **Research Q3:** In terms of setting up the CI/CD pipeline how easy it is to set up the CI/CD workflow with the existing development components?
- **Research Q4:** How can modular CI/CD pipeline architectures make Industrial IoT systems work together, be more reliable, and be more automated?
- **Research Q5:** How can automated reporting systems help people who are part of the CI/CD process see things more clearly and make better choices?

1.4 Methodology

In order to achieve the aforementioned objectives, the gaps will be studied extensively in the existing literature review. The suitable tools considered as a result will then be selected.

For the implementation, we will consider an existing infrastructure in the development environment that has been deployed manually using Terraform CDK which will be used as project to build the pipeline.

After the successful completion of the pipelines, we will compare the results of the metrics (build time, usability etc.), and conclude our research.

1.5 Limitation of Study

Although this study provides a comparative evaluation of CI/CD platforms is within the Industrial IoT domain, several technical limitations should be acknowledged.

Firstly, the existing experimental application is based in JavaScript in an AWS-environment therefore the research may not provide an extensive comparative analysis how the CI/CD pipeline tools might behave with other cloud provider environments.

Secondly, the experimental setup exposes another limitation. The programming language used to develop using a JavaScript ecosystem. Applications developed using

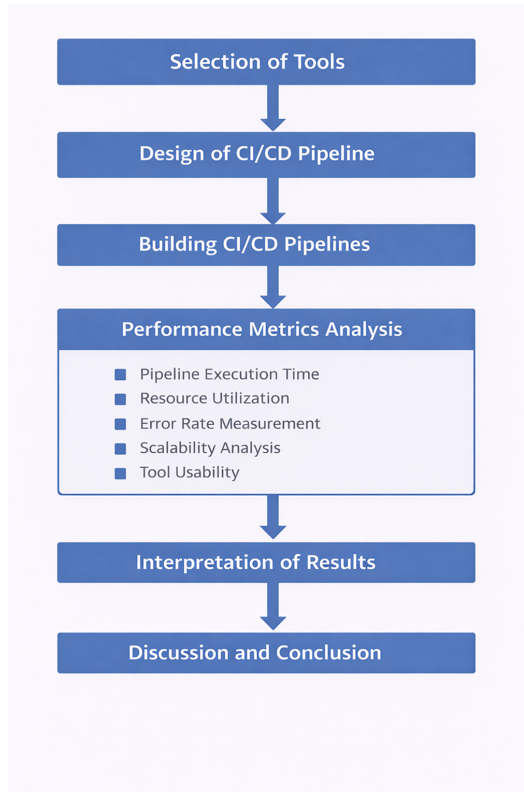

Figure 1.1: Methodological Workflow for CI/CD Pipeline Evaluation

other programming languages might produce different results due to factors such as: testing workflows, dependency management, and build processes.

Lastly, for the purpose of maintaining cost effectiveness minimum resources are allocated to instance size used in the experimental architecture which can also contribute to performance issues and can influence pipeline execution times.

Lastly, external factors, such as network latency and resource contention, could also influence the results.

1.6 Thesis Organization

The thesis work has been organised in the following arrangement:

- **Chapter 1:** Gives an introduction about problem statement, need of study, objectives of the research, and methodology.
- **Chapter 2:** Contains relevant literature review about the CI/CD pipelines.
- **Chapter 3:** Includes the design, description, and implementation of the CI/CD pipeline.
- **Chapter 4:** Explores, discusses and interprets the obtained results.
- **Chapter 5:** Incorporates the conclusion and recommendation for further study.

Chapter 2

Literature Review

In the DevOps landscape, the reliability and speed of modern software development have been improved by the adoption of CI/CD pipelines. As new tools emerge in the marketplace they continue to streamline the software build, test and deployment processes.

Historically, manual deployment required human intervention: meaning it required more coordinated efforts from multiple cross-functional team members for efforts as little as software release in staging environment. Releasing software to production environment was even more complicated and time-consuming time. Many organization employed multiple steps in their deployment process which made the manual deployment error-prone, and difficult to repeat consistently [3].

With the adoption of automated deployment through CI/CD pipeline, deployment errors can be significantly reduced. According to one study, comparing manual deployment with automated deployment, shows that CI/CD pipeline can reduce deployment errors by 85 percent by removing human intervention and ensuring consistent execution of deployment steps [4].

The execution of repeatable deployment pipeline manages the sequence of steps: build, test, deploy, and release. Automation has enabled the execution of routine tasks without manual intervention which enables developers trigger complex deployment processes through automated system instead of deploying each step manually [3].

Automated deployment has reduced the time of deployment processed enabling faster releases to end-users, which is an important software deployment goal. The value of software is determined by it's real operational value. Empirical studies suggest that CI/CD pipeline reduces deployment 60-70 percent mainly by eliminating manual deployment and repetitive deployment steps[4].

Within the DevOps principles, the adoption of unique features of CI/CD tools such as Jenkins and Gitlab CI has gained significant attention with their usability in diverse environments. In this section we will be exploring the existing studies done on these tools and focus on their strengths limitation and suitability in different scenarios.

2.0.1 Evolution of CI/CD and Industry Adoption of DevOps Methdologies

As applications continue to become larger and more distributed the adoption of automated deployment practices that initially segregated operational and development teams reveals clear limitations. It became increasingly difficult to maintain consistency in deployment cycles which further delayed the release cycle and feedback. As a result the DevOps movement emerged to address these bottlenecks as a collaborative effort to improve communication, automation, and efficiency through the software development cycles. Research by Forgsen et al. indicates that high-peforming DevOps team deploy software 208 times faster than low-performing teams with the adoption of automated deployment processes [5].

Within this context CI/CD practices evolved and became the integral component of DevOps. Continuous Integration focuses on the frequent integration of code changes into shared repositories while aiding in recognizing integration issues early in the development phase. Continuous Delivery and Continuous Deployment take this approach a step further by automating the next steps in the software lifecycle, such as testing, packaging, and putting the software into production environments. These practices work together to help companies deliver software updates more often while keeping the system stable and reliable [6].

The DevOps culture strongly supports the use of CI/CD practices. This culture stresses the communicative collaboration between development and operations teams to deliver applications rapidly, and efficiently while ensuring system stability and reliability [7]. As more companies aim to automate the process of building, testing, and deploying software, many different CI/CD tools have emerged to help with the adoption of CI/CD. For example, GitLab CI, and Jenkins are some of the most popular tools for managing modern software delivery pipelines. They all have different features that make them useful.

2.0.1.1 Jenkins

Jenkins is one of the legacy CI/CD tools. It has a vast ecosystem of plugins. Even though Jenkins has a steep learning curve initially but it is widely used within the industry.

2.0.1.2 Gitlab CI

Integrated within the Gitlab platform it simplifies workflows by combining version control, issue tracking, and deployment capabilities within one single platform.

2.0.2 Challenges in CI/CD Implementation

There are many benefits to CI/CD pipelines, however, there are also challenges as well. For example, they can be difficult to set up and keep running. The high initial costs and the difficulty of setting up automated pipelines can be complicated for

smaller development teams. Moreover, it's also futile and often difficult for companies to compare CI/CD tools and make credible choices as there aren't any standard criteria to measure their performance. However, recent trends in CI/CD development are more focused on making it easier to set up workflows, making systems more scalable, and adding stealthier security features for improved support of modern software development methods. A large empirical study analyzing 84,475 Android open-source projects discovered that about 10 percent projects use CI/CD tools and services, which means that 90 percent projects didn't adopt CI/CD pipelines and still relies of traditional development workflows [8]. Several studies highlight barriers to adoption, including organizational resistance, high implementation complexity, and lack of automation maturity [6].

2.1 CI/CD in Distributed and Cloud-Native Systems

According to one statistics by the European Union, about 52.74 percent of modern software systems in 2025, rely distributed and cloud-native architectures. Cloud-native Architecture support pay-as-you go scalability, enhanced flexibility, and high availability. These architectures typically consist of multiple combined services deployed across cloud infrastructure, edge devices, and distributed computing environments. In such systems, software components are maintained and updated autonomously and infrastructure resources must be dynamically provisioned and managed [9]. As a result, automated software delivery mechanisms such as CI/CD pipelines have become the industrial standard for managing the complexity associated with these environments [3].

Considering Internet of Things and Industrial IoT platforms, the need for automated delivery pipelines becomes even more critical. IoT systems typically involve a combination of gateways, sensors, edge devices, and cloud micro-services allowing continuous data-flow across distributed networks. For example, smart home systems and environmental monitoring platforms is often based on cloud-based microservices to collect and process data generated by connected devices. Therefore, managing updates to these services requires consistent testing, integration, and deployment processes to ensure that device communication and data processing workflows remain stable. Industrial IoT systems exposed an additional level of complexity due to the integration of operational technologies with microservices-based cloud computing infrastructure [10]. Software services are responsible for the ingestion of large volumes of telemetry data from machines and industrial sensors in industrial environments such as smart manufacturing, maintenance systems, and industrial asset monitoring platforms [11]. These services often run within and parallel to cloud-native microservice architectures that support real-time analytics, data processing, and monitoring. Because these systems operate in critical environments, updates must be carefully validated to avoid disruptions to operational processes [12]. Through CI/CD pipeline companies are able to manage these complex distributed systems by automating integration, streamlined testing, and deployment workflows. Development teams are able

to validate application code, infrastructure configurations, and service interactions before changes are deployed to production environments, through the automated pipelines. Within the IoT and IIoT domains, CI/CD pipeline helps ensure that patch or infrastructure updates to cloud services, data-ingestion pipelines, while supporting infrastructure that can be deployed reliably while minimizing the risk of operational failures [10]. Another critical advantage of CI/CD in distributed systems is the ability to support microservice-based architectures in cloud-native environments [13]. Microservices allow IoT platforms to segregate different parts of software into smaller services meant for specific tasks such as device communication, data ingestion, event processing, and analytics. CI/CD pipelines allow each of these micro-services to be developed, tested, and deployed independently while maintaining overall system reliability and stability [14].

In summary, the growing dependence on cloud-native based distributed architecture has transformed the way modern software systems are developed and deployed, making it a standard in DevOps. This change is particularly evident in Industrial IoT environments, where applications must operate across diverse infrastructures consisting of edge devices, cloud services, and data-processing platforms but independent self-contained services within a distributed system. Operating and managing such complex systems requires reliable mechanisms to coordinate software updates, infrastructure changes, and service interactions while striking a balance between operational stability and infrastructure reliability [13].

By adopting modular microservice architectures in Industrial IoT and automated infrastructure validation workflows, testing, and deployment, CI/CD pipelines help organizations manage the increasing complexity of different independent components of Industrial IoT based distributed architecture while maintaining system reliability and development agility as a whole. Consequently, understanding how CI/CD platforms perform in these environments has become an important area of research, particularly in the context of Industrial IoT systems where deployment reliability and operational continuity are critical [13] [15].

2.2 Performance Metrics in CI/CD

To evaluate the performance and the effectiveness of CI/CD pipelines a measurable performance indicators is required that reflect how efficiently automated workflows operate within a end-to-end software delivery process. It is crucial to define quantitative metrics to assess the reliability, efficiency, and scalability of their automated deployment processes. These performance metrics provide indicators into how well CI/CD platforms behave and manage integration, testing, and deployment activities in complex distributed development environments especially involving cloud-native architectures.

To evaluate the effectiveness and performance of CI/CD pipelines a measurable performance indicators is required that reflect how efficiently automated workflows operate within a end-to-end software delivery process. It is crucial to define quanti-

tative metrics to assess the reliability, efficiency, and scalability of their automated deployment processes. These performance metrics provide indicators into how well CI/CD platforms behave and manage integration, testing, and deployment activities in complex distributed development environments especially involving cloud-native architectures.

2.2.0.1 Execution Time

In the performance evaluation of CI/CD pipeline, the execution time is the widely used metric. Execution time refers to the complete time required to complete the entire pipeline, including the build, test, deployment phases, code quality analysis, and reporting. Shorter pipeline execution times provide quicker feedback to development teams and other stakeholders involved, making it possible to spot and fix problems sooner in the development cycle. Rapid feedback loops are especially valuable in settings where numerous cross-functional teams regularly integrate code updates into shared repositories.

2.2.0.2 Resource Utilization

Resource utilization measures the consumption of computational resources such as CPU, memory, and storage during pipeline execution. Efficient resource utilization ensures that CI/CD pipelines do not cause any unnecessary financial overhead on top of the underlying infrastructure.

2.2.0.3 CPU and Memory Usage

High resource consumption such as CPU and memory usage can negatively impact the performance of CI/CD pipelines. Especially, when multiple applications share the same computing environment. Efficient resource utilization is required necessity to ensure that automated pipelines do not overload the infrastructure or increase operational costs. In Industrial IoT environments where large volume of data is involved therefore with high read and write operations, tools may experience performance bottlenecks that slow down pipeline execution. Resource-intensive CI/CD platforms may perform well in isolated environments but can face difficulties in shared or cloud-based infrastructures. As a result, organizations often need to balance performance and operational costs when deploying pipelines on cloud platforms such as AWS. In terms of tool-specific observations, GitLab CI has been shown to demonstrate efficient resource utilization in containerized environments through its integration with Kubernetes and Docker, whereas Jenkins may require careful configuration and optimization to avoid excessive resource consumption, especially when handling multiple concurrent pipelines.

2.2.0.4 Error Rates

While managing complex automated workflows error rates are commonly used as indicators of the reliability and efficiency of CI/CD pipelines. Different types of failures may occur during the execution of a pipeline, each caused by a different error.

- Build failures can arise from issues related to dependency resolution or errors during code compilation.
- Test failures may occur during the automated execution of test cases, often due to incompatible frameworks, configuration errors, or missing dependencies.
- Other errors may also appear in later stages of the pipeline, including packaging, staging, or deployment before the application is released to production.

The following two formulas are used to evaluate the reliability of a CI/CD pipeline.

The **pipeline error** rate can be measured as total number of failed pipelines with the total number of pipeline executions. This metric provides an indication of how frequently pipelines fail during the software delivery process and can be expressed as:

$$\text{Pipeline Error Rate} = \frac{\text{Number of Failed Pipelines}}{\text{Total Number of Pipelines Executed}} \times 100 \tag{2.1}$$

The **job error rate** measures the number of failed jobs relative over the total number of executed jobs, as shown in:

$$\text{Job Error Rate} = \frac{\text{Number of Failed Jobs}}{\text{Total Number of Jobs Executed}} \times 100 \tag{2.2}$$

2.2.0.5 Tool Usability

Tool usability can often be deemed as one of the simplest factor considered in this comparative analysis however, it still has impact on the the overall set up causing operational overhead. Tools that are difficult to configure and manage while requiring complex initial setup may introduce development challenges for the development teams. Limited usability or poorly designed platforms can cause delays in troubleshooting or adjusting pipeline configurations. Therefore usability is considered an important factor while comparing tools.

2.2.0.6 Scalability

Scalability is an important factor when existing infrastructure becomes more complex, therefore, the tool used to develop the CI/CD pipeline must support the complex pipelines. Referring to the experimental architecture that involves multiple stages. Therefore, in such cases when CI/CD pipelines are deployed in cloud environments, effective scaling strategies are necessary to maintain performance while controlling operational costs. Studies examining pipeline performance under load have shown that CI/CD tools differ in their ability to scale efficiently.

- GitLab CI, supports parallel pipeline execution and integrates well with container orchestration platforms, enabling it to handle concurrent workloads more effectively in many complex cases.
- However, Jenkins is highly flexible due its extensive plugin ecosystem. Even though it requires additional configuration but there is a high flexibility of customization according to the DevOps requirement.

2.3 Selection of Tools

The selection of appropriate CI/CD platforms is a critical step in evaluating automated SDLC. Currently, there is a wide variety of CI/CD tools, each with it's own benefits and sets of challenges. However, for this experimental setup we have chosen two types of CI/CD tools: one that is open-source with an extensive plugin ecosystem, and another that is a centralized platform. Therefore, Jenkins and GitLab CI/CD were selected for this study due to their widespread adoption, while considering strong community support due to this experiment's complex pipeline, and suitability for complex DevOps environments.

Jenkins is considered as one of the most established open-source CI/CD platforms within the modern software industry. It is offering extensive flexibility through its large plugin ecosystem and ability to support highly customizable pipelines. On the contrary, Gitlab CI, represents a more modern integrated approach, providing a centralized platform that combines version control, CI/CD automation, and monitoring capabilities. Both tools are widely used in enterprise environments and support distributed and cloud-native architectures, making them suitable candidates for evaluating CI/CD performance in modern software development workflows. While other CI/CD platforms such as GitHub Actions, CircleCI, and Travis CI also provide automation capabilities, they were not included in this study in order to maintain a focused comparison between two mature and widely adopted CI/CD solutions with different architectural approaches to pipeline orchestration.

2.4 Comparison of Selected Tools

2.4.0.1 Jenkins: A Legacy CI/CD Tool

Released in 2004, Jenkins is an open-source automation server that has played a crucial role in molding the landscape of automated deployments. It is one of the most widely used CI/CD platforms due to its flexibility, and extensive plugin ecosystem enabling high customization [16]. It supports 1,500 plugins making it to be a highly modular architecture, and therefore enabling integration with a wide range of third-party tools, programming languages, frameworks, and deployment environments [16]. With this variety Jenkins is able to support diverse software deployment patterns, ranging from simple build automation to complex multi-stage deployment pipelines.

Apart from these strengths, Jenkins' architecture exposes certain limitations, especially in large-scale or cloud-native environments. Since Jenkins relies heavily on plugins for its functionality, therefore maintaining compatibility between plugins and managing frequent updates can require significant operational effort [17]. Moreover, the adoption of CI/CD through Jenkins architecture can require precise configuration while resource management to scale efficiently in distributed environments, becomes increasingly hard.

Jenkins, however, still continues to remain a widely adopted CI/CD tool, largely due to its maintained presence in the DevOps ecosystem, strong community support, and it staunch reputation for it's reliability. But since it's an open-source software there is no customer support despite having a wife community support. Moreover, even though Jenkins' flexibility allows extensive customization in enterprise-level CI/CD workflows, such as complex microservices-based architectures but sometimes it can become a operational bottleneck for smaller teams with limited DevOps capacity [17].

2.4.0.2 Gitlab CI: A centralised DevOps Ecosystem

GitLab CI/CD is tightly integrated into the GitLab platform and provides a unified environment that combines version control, CI/CD functionality, and monitoring capabilities. This integration eliminates the need for multiple external tools, simplifying the configuration and management of CI/CD workflows. GitLab CI uses a YAML-based configuration system in which pipeline policies and workflows are defined. Through this configuration, developers can specify pipeline stages, job dependencies, and conditional execution rules, allowing for flexible and structured automation processes [18].

GitLab CI/CD is tightly integrated into the GitLab platform and provides a unified environment that combines version control, CI/CD functionality, and monitoring capabilities[18]. This integration eliminates the need for multiple external tools, simplifying the configuration and management of CI/CD workflows. GitLab CI uses a YAML-based configuration system in which pipeline policies and workflows are defined. Through this configuration, developers can specify pipeline stages, job dependencies, and conditional execution rules, allowing for flexible and structured automation processes [18].

One of the eminent strengths of GitLab CI is its strong support and compatibility with the containerization technologies e.g Docker and Kubernetes. This feature makes the platform suitable for modular cloud-native applications relying on container-based deployment models. For example, later discussed in Chapter 3, our design integrates Sonarqube using docker and Gitlab CI provided smooth integration. Additionally, there is built-in security features in GitLab CI such as sast/dast and compliance features, such as vulnerability scanning and dependency checks, which help organizations address the growing demand for secure software delivery practices [18].

In several comparative studies GitLab CI's advantages in terms of ease of setup

and scalability when compared with traditional CI/CD tools has been highlighted. Its ability to manage concurrent pipelines and run parallel jobs efficiently with relatively minimal configuration adjustments makes it well suited for large-scale enterprise level projects with modular architecture. However, apart from these advantages, GitLab CI can have a learning curve for teams unfamiliar with its configuration model, but due to their well maintained documentation and community support it's easy to figure it out. In some cases, its resource consumption and the complexity of advanced pipeline configurations may pose challenges for teams transitioning from simpler CI/CD solutions.

2.5 Comparative Analysis and Challenges

When Jenkins and Gitlab CI are compared across important performance metrics, it reveals valuable insights into the strengths and limitations. And it becomes evident that these two tools differ from one and another significantly, especially in reliability, resource utilization, usability, and scalability. However, comparing error rates poses some limitations which is discussed thoroughly in the next section.

Jenkins stands out in highly customizable architecture with it's vast ecosystem of plugins that enable teams to create complex pipelines that are customized to their specific requirements [17]. Even though, the default configuration of the Jenkins pipeline may lead to delayed build times if it is not optimized efficiently. Causing delays due to improperly configured plugins or inefficient pipeline stages can be a possibility if not considered in the initial stages of adopting DevOps practices. Therefore, designing the pipeline while keeping optimization in mind becomes crucial Jenkins has the potential to attain performance that is comparable to other CI/CD platforms with the implementation of suitable configuration and optimization strategies. Due to it's concurrent job execution, pipeline caching, and built-in support GitLab CI repeatedly exhibits efficient build execution in large-scale environments. Consequently, it is well-suited for projects that requires frequent deployments [3].

Jenkins and Gitlab CI have some distinct differences when it comes to error detection. Jenkins has a plugin-centric architecture where Gitlab CI has a centralized architecture. Due to the extensive ecosystem of plugins, custom error detection and monitoring mechanisms can be incorporated into Jenkins [17]. Although this flexibility is beneficial for detailed and complex workflows, it may include additional configuration complexity for teams. On the other hand, due to it's centralized nature of platform, GitLab CI provides a variety of built-in mechanisms for pipeline monitoring and error detection, notably in containerized and Kubernetes-based environments. These features include vulnerability scanning and dependency checks. Which means there is no additional technical overhead for the development team to try to figure out which plugins to use, instead the Gitlab CI already provides these features.

Both GitLab CI and Jenkins can become resource-intensive when administering

complex pipelines or high workloads in terms of resource utilization. Therefore, it is highly important to optimize pipelines and allocate resources efficiently in order to preserve consistent performance in large-scale distributed development environments such as in Industrial IoT. From the perspective of usability, GitLab CI offers a more streamlined and centralized configuration process due to its integrated and centralized development platform and YAML-based pipeline definitions [18]. In comparison, the plugin-centric architecture and manual configuration requirements of Jenkins, despite its high flexibility, typically result in a more challenging learning curve and requires human intervention therefore causing operational and developmental overhead.

Scalability is another salient factor that should be considered when designing CI/CD pipelines that operate in cloud-native and distributed environments such as Industrial IoT base architecture. Especially, when integrated with container orchestration platforms, in our case we employ the use of docker to integrate Sonarqube, GitLab CI has exhibited robust capabilities in managing large-scale deployments and concurrent pipelines even when it's intricate and complex [18]. Jenkins structure makes it capable of accommodating scalable workflows; however, it frequently necessitates additional infrastructure administration and configuration to ensure optimal performance in cloud-native environments.

Despite these factors, there are still challenges faced when conducting a comparative analysis. For example, one evident constraint is the absence of standardized benchmarking methodologies or criteria for the assessment of performance of these CI/CD tools, consequently helping in tool selection. Moreover, performance results can be significantly affected by even small variations in infrastructure environments, pipeline complexity, conflicting configurations, testing frameworks, and application size and complexity, which is widely considered during our experimental setup. In addition, the rapid evolution of CI/CD platforms necessitates ongoing evaluation, as they introduce new features and architectural enhancements.

2.6 Research Gap in Industrial IoT CI/CD and Future Implications

Cue to the scarcity of empirical studies that assess CI/CD platforms in contemporary cloud-native environments, there is a research gap in comparison of CI/CD platforms and consequently there implications in the Industrial IoT domain. There is a growing demand for research that investigates the performance of CI/CD tools under these conditions as container orchestration technologies and distributed microservice architectures continue to expand. Systematic experimentation and empirical evaluation are necessary to address these challenges and gain a more comprehensive understanding of the ways in which various CI/CD platforms facilitate scalable and reliable software delivery processes.

Chapter 3

Pipeline Design and Setup

3.1 Infrastructure Architecture Review

To evaluate the performance and operational characteristics of different CI/CD platforms, it was necessary to design an infrastructure environment that reflects the complexity of modern distributed systems. The infrastructure used in this study was intentionally designed as a multi-component cloud-native architecture in order to simulate realistic deployment conditions typically observed in Industrial IoT and data-driven microservice platforms.

tributing system functionality across several independent services and infrastructure layers, the architecture enables detailed testing of CI/CD pipelines under conditions that involve multiple dependencies, asynchronous communication, and infrastructure provisioning.

The architecture shown in Figure 3.1 illustrates a cloud-based environment deployed within a virtual private network where application services, data processing components, and communication interfaces operate as loosely coupled modules. The infrastructure integrates several cloud services including message brokers, event-driven compute functions, databases, storage services, and API gateways. This distributed design allows each component to operate independently while still participating in a coordinated data processing workflow.

One of the primary motivations behind designing this architecture was to intentionally increase the complexity of the deployment pipeline. A more intricate infrastructure allows the CI/CD workflow to incorporate multiple stages such as infrastructure validation, automated testing, service integration checks, and deployment orchestration. This approach makes it possible to design comprehensive test suites that evaluate not only application code but also infrastructure configurations and interactions between system components. As a result, the CI/CD pipelines developed in this study are capable of validating different aspects of the system, including service communication, infrastructure provisioning, and deployment consistency.

Furthermore, separating system components into independent services allows the pipeline to test them in isolation as well as within the broader system environment. Due to it's complex distributed design involving multiple independent microservices,

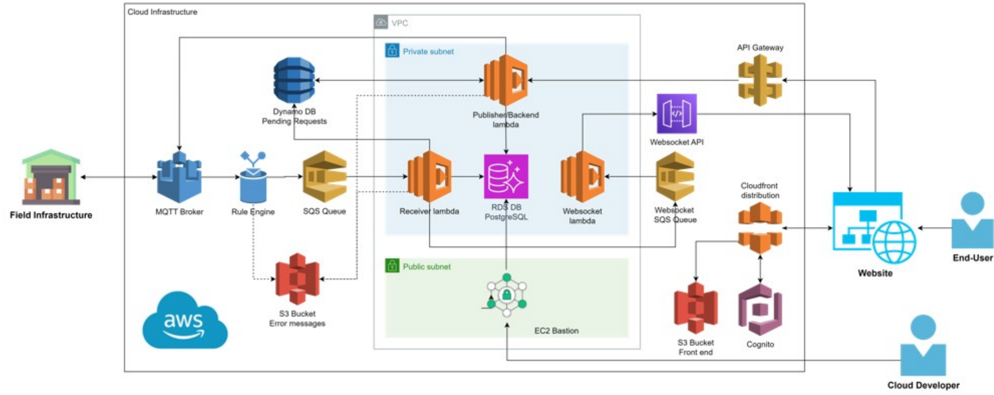

Figure 3.1: Experimental Infrastructure for CI/CD Evaluation

this modular software design is especially useful for evaluating CI/CD platforms as it introduces realistic operational scenarios such as parallel deployments, asynchronous message processing, and dependency management across services. These characteristics provide a suitable foundation for performing a comparative analysis of CI/CD tools, as they allow the pipelines to exercise different aspects of automation, scalability, and orchestration capabilities.

The following sections describe the architecture in more detail, starting with the design of the CI/CD pipeline and the rationale behind the selected testing stages. Subsequent sections then explain how the infrastructure and pipeline configurations were implemented to support the experimental evaluation.

3.2 System Architecture Components

The architecture in figure 3.1 represents all the interconnected microservices that is responsible for handling the data generation, transmission, and transformation. In real life scenario in the Industrial IoT setup these devices generate data, which is then transmitted as telemetry messages to the cloud infrastructure as API requests. These telemetry messages are the starting point of data processing lifecycles and are further processed through multiple stages as part of the ETL pipelines, before the final data are exposed in the front-end in a desired manner to the end users. Moreover, all these interconnected components operate independently of each other, while remaining integrated within the overall system. The infrastucture used in this study is a replica of the production environment to simulate real-world data ingestion scenarios typically found in Industrial IoT environments. The infrastructure was provisioned using Terraform CDK, in the form of CloudFormation templates, and these templates were deployed manually during the initial setup locally through **cdktf** command.

3.2.0.1 Field Infrastructure

Field infrastructure represents the external IoT devices such as edge devices, gateways, sensors etc. that are generating operational data. These devices transmit telemetry messages through API request to the cloud infrastructure for further processing.

3.2.0.2 MQTT Broker

It receives telemetry data from field infrastructure and forwards it to the internal cloud services for processing therefore acting as the entry point. MQTT is mainly used in Industrial IoT systems due to its lightweight messaging protocol and support for asynchronous communication during data reception.

3.2.0.3 Rule Engine

In the rule engine the incoming messages are processed and determines how they should be routed within the system meaning it could be disregarded as a duplicate, false positives etc. Based on predefined rules based on certain thresholds, messages can be forwarded to downstream processing services or stored for further analysis or disregarded. This component enables flexible routing and event-driven processing within the architecture.

3.2.0.4 Message Queue (SQS)

The queue is used to distinguish message ingestion from backend processing components. Messages placed in the queue can be processed asynchronously by compute services, which improves system scalability and reliability, e.g. message can have two different timestamps: receivedAt vs. createdAt, showing the time taken to process and ingest the data. The queue also allows the architecture to handle bursts of incoming data without overwhelming processing services which makes this infrastructure robust to large data handling.

3.2.0.5 Serverless Processing Functions

There are multiple serverless processing functions involved. Here the serverless functions or lambda functions are responsible for processing messages retrieved from the queue, acting as a microservice. Each data source may have it's own serverless function. These functions perform tasks such as data transformation, validation, and preparation before storing the processed data in the database and before it is fed to the ingestor it is first sent to the Message SQS. The event-driven nature of these functions allows the system to scale dynamically based on workload.

3.2.0.6 Database Layer

A relational database such as Aurora cluster is used to store structured data generated by the processing functions. In this experimental setup the database used had minimum resources assigned to ensure low cost. The database supports queries

from backend services and enables persistent storage of processed information which is later utilized in the front-end. Including a database component introduces an infrastructure dependency that must be validated during deployment.

3.2.0.7 WebSocket Communication Layer

The WebSocket layer enables real-time communication between backend services and the frontend application, it connects the private VPC subnet and public VPC subnet. This allows the system to push updates to connected users without requiring continuous polling requests and compromising security at the same time. The inclusion of WebSocket services supports applications that require real-time data visualization which is a key feature in the Industrial IoT implementation as users want to see real-time data in the front-end to make informed decisions.

3.2.0.8 API Gateway

The API Gateway provides an interface between external clients e.g the data received through API requests from the IoT devices and the backend services. It manages incoming requests and routes them to appropriate services while also handling authentication and request validation. This component enables secure and structured communication with external users.

3.2.0.9 Frontend Application and Content Delivery

The frontend application provides the user interface a centralized platform where they can see the entire data being received through these devices allowing end users to interact with the system and make informed and strategic decision. Static content is hosted in cloud storage and distributed through a content delivery network to improve performance and reduce latency.

3.2.0.10 Developer Access and Management

Administrative access mechanisms allow developers to manage and monitor the infrastructure environment. For example in our design a bastion host has been developed to access the backend infrastructure securely through ssh. These tools enable debugging, system maintenance, and infrastructure management during development and testing.

3.3 Pipeline Design and Architecture

3.3.0.1 CI/CD Pipeline Workflow

The CI/CD pipeline architecture illustrated in Figure 3.2 describes the automated workflow used to build, test, deploy, and evaluate the application while generating automated reporting for stakeholders. The process begins when source code is pushed to the Gitlab repository, which triggers the CI/CD platform implemented using

Jenkins or GitLab CI in our experimental setup. The pipeline first executes the build stage, followed by automated unit testing using the Mocha testing framework which is written for each stack.

3.3.0.2 Sonarqube Integration

After successful or partially successful unit tests, the pipeline performs static code quality analysis using SonarQube to evaluate code quality and detect potential issues before deployment. To incorporate automated code quality assessment within the CI/CD workflow, SonarQube was integrated as a static code analysis tool. The SonarQube Community Edition was deployed as a Docker container, enabling a lightweight and portable deployment within the experimental environment. During pipeline execution, the SonarQube scanner was triggered to analyze the source code and detect issues such as code smells, potential bugs, and security vulnerabilities. The analysis results were uploaded to the SonarQube server and visualized through its dashboard, providing continuous feedback on code quality and helping maintain coding standards across the project.

3.3.0.3 Unit and E2E testing

One important design consideration that was incorporated after conducting several experimental iterations was the handling of test failures within the CI/CD pipeline. Rather than terminating the pipeline immediately when a unit or end-to-end (E2E) test failed, the pipeline was configured to continue execution while collecting and reporting the results of all testing stages. This design choice ensured that complete diagnostic information was captured and included in the final pipeline report, allowing developers to analyze failures more effectively without interrupting the entire deployment workflow.

To support this approach, automated test suites were implemented for each infrastructure stack deployed in the system. Unit tests were used to validate individual components and functions during the early stages of the pipeline, while end-to-end (E2E) tests were designed to verify the correct provisioning and behavior of deployed cloud resources. For example, in the case of the request storage stack, automated tests were implemented to verify the existence and operational status of critical infrastructure components such as DynamoDB tables. These tests used the AWS SDK to query deployed resources and confirm that they were correctly provisioned and in an active state. Once these validation stages were completed, the application was deployed to the target platform where additional regression and E2E tests were executed. These tests validated the overall system behavior in a runtime environment and ensured that interactions between infrastructure components, application services, and external dependencies functioned as expected. By combining infrastructure validation, unit testing, and runtime E2E verification, the pipeline provided a comprehensive testing strategy that supported reliable and repeatable deployments in the experimental environment.

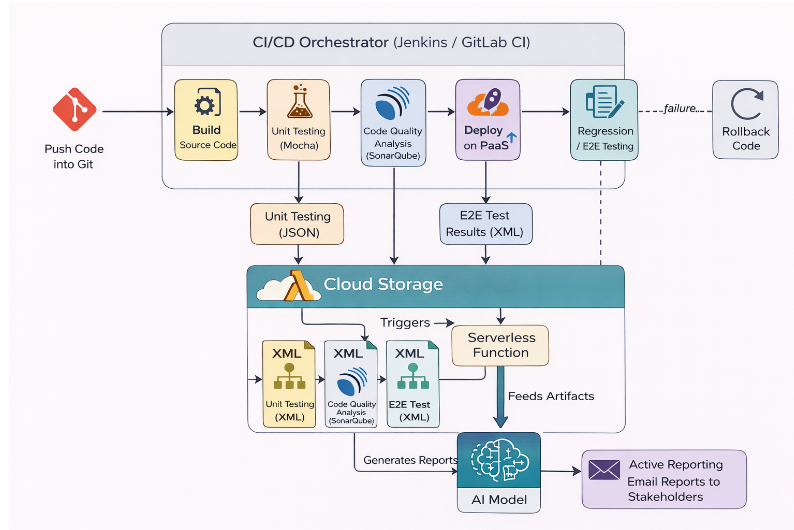

Figure 3.2: AWS based Distributed Architecture

3.3.0.4 Serverless Reporting System

To improve the interpretability of pipeline outcomes and reduce the need for manual inspection of raw test artifacts, a serverless reporting system was designed and integrated into the CI/CD workflow. During pipeline execution, the outputs of the validation stages, including unit testing results, code quality analysis results, and end-to-end (E2E) testing results, are generated as artifacts in XML and log-based formats. These artifacts are uploaded to an Amazon S3 bucket, where they are organized into dedicated folders according to execution context. In this setup, the S3 bucket acts as a centralized repository for pipeline outputs and serves as the entry point for the reporting workflow.

Whenever new artifacts are uploaded, Amazon S3 generates an event notification that is forwarded through Amazon SNS, which in turn triggers an AWS Lambda function responsible for processing the new results. The Lambda function, implemented in Node.js, retrieves the uploaded test and analysis artifacts from the S3 bucket, aggregates the relevant files for a given pipeline run, and prepares the input for AI-based summarization. In particular, the function collects unit test outputs, E2E test outputs, and SonarQube analysis logs before consolidating them into a single structured input payload.

The AI model used in this experimental setup was the Amazon Bedrock foundation model **verb|amazon.titan-text-express-v1**|. This model was selected primarily on the basis of simplicity and cost-effectiveness, making it suitable for generating concise reporting summaries without introducing unnecessary architectural complexity. The Lambda function invokes the Bedrock runtime through the
verb|InvokeModelCommand| API and supplies a prompt specifically designed for CI/CD reporting. The prompt instructs the model to act as an CI quality assistant and to generate a clear, executive-friendly report based on the raw pipeline outputs. More specifically, the model is asked to include sections covering the overall

build and test overview, unit and integration test summaries, code quality issues identified by SonarQube, notable risks or trends, and recommended action items for the development team. The prompt also explicitly instructs the model to remain concise and to state assumptions whenever data are incomplete. The following prompt was used fed into the AI model while generating

You are a CI reporting assistant. Produce ONE concise, email-ready report that covers ALL stacks. MUST follow this outline:

- *Summary (3–5 bullets)*
- *Per-Stack Results (for each stack in alphabetical order):*
  - *Tests: pass/fail counts and notable failures (if any)*
  - *Code Quality: key rules/severities from SonarQube (if any)*
- *Risks (bullets)*
- *Action Items (bullets)*

Rules:

- *Be factual and non-repetitive.*

- *If a section has no data, state "no data".*

- *Keep total under 500 words.*

- *No marketing language.*

- *Always Include this text at the end of the email "Review all the test results here: s3://banu-pipeline-reports/test-results/*

An additional consideration in the design of the serverless reporting system was operational cost. Since the reporting workflow is executed only when pipeline artifacts are generated, the system follows a pay-as-you-go pricing model. The primary cost component is the invocation of the Amazon Bedrock model **amazon.titan-text-express-v1**, which is priced based on the number of input and output tokens processed during text generation. For the experimental setup, each report typically consumed approximately 1,000–1,500 input tokens derived from test artifacts and generated around 400–800 output tokens in the final summarized report. Based on current Amazon Bedrock pricing, this results in an estimated cost of approximately $0.002–$0.005 per report generation.

Additional infrastructure costs are minimal. AWS Lambda executions fall well within the free tier for the small compute duration required for processing artifacts, while Amazon SNS notifications and Amazon SES email deliveries incur negligible charges at the scale of the experiments. Storage costs in Amazon S3 are also minimal since the artifacts consist primarily of small XML files and log outputs. As a result, the overall operational cost of the automated reporting system remains extremely low, making the solution both scalable and economically feasible for continuous CI/CD monitoring. Once the report has been generated, it is distributed automatically to

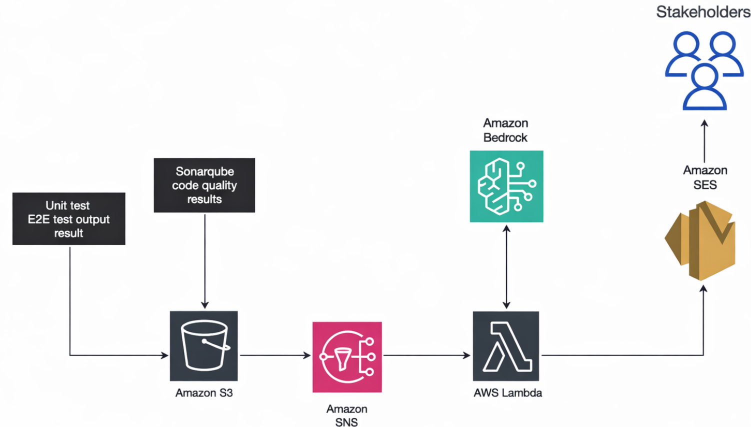

Figure 3.3: Serverless Reporting System

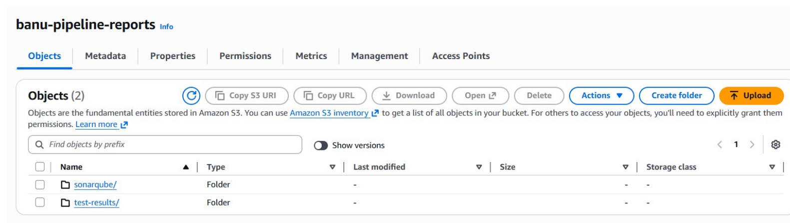

Figure 3.4: Folder Structure in the S3 bucket

stakeholders using Amazon SES. Email recipients are configured through environment variables, and the generated report is sent as a plain-text email summary. This allows both technical and non-technical stakeholders to receive an immediate overview of the pipeline results without having to inspect raw logs or XML artifacts manually. In addition to the AI-generated summary, the reporting workflow also provides access to the original artifacts stored in S3, enabling developers to review detailed test outputs and SonarQube analysis when further investigation is required.

This serverless design offers several advantages within the CI/CD workflow. First, it separates artifact analysis from the main pipeline execution, preventing reporting logic from increasing build and deployment times. Second, the event-driven nature of the design ensures that reporting is triggered automatically whenever new artifacts are produced. Finally, by combining cloud storage, event notifications, serverless computation, AI-based summarization, and automated email distribution, the reporting system enhances stakeholder visibility while preserving the scalability and modularity of the overall pipeline architecture.

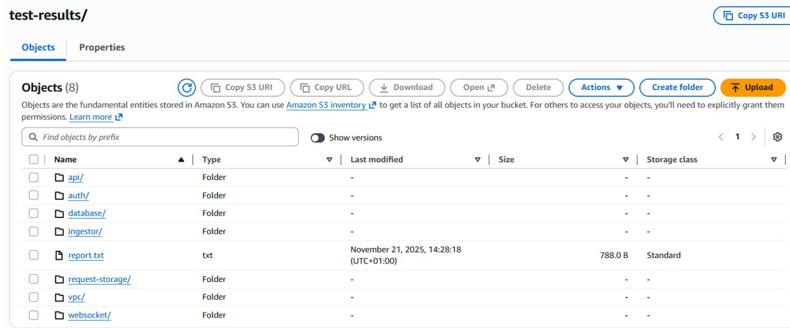

Figure 3.5: Test Results of Each Stack

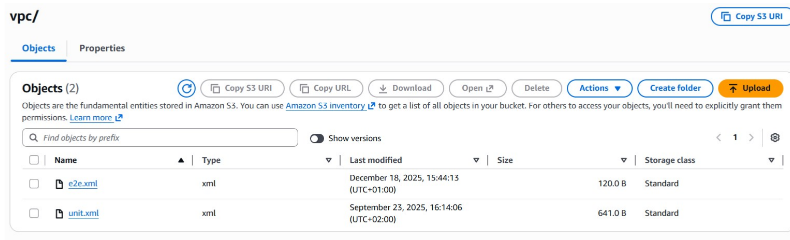

Figure 3.6: .XML file of VPC Stack

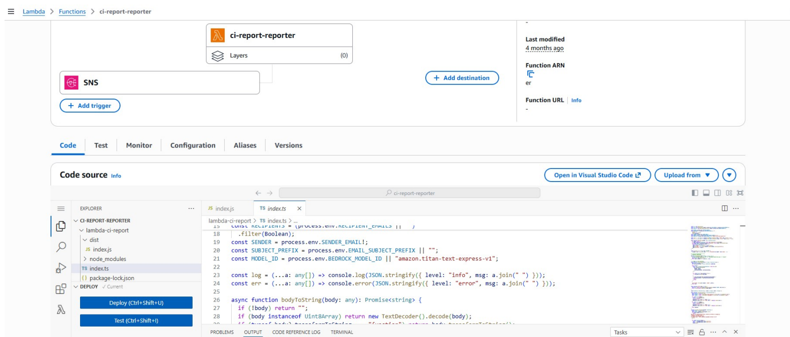

Figure 3.7: Serverless AI Reporting Stack Deployed using Terraform

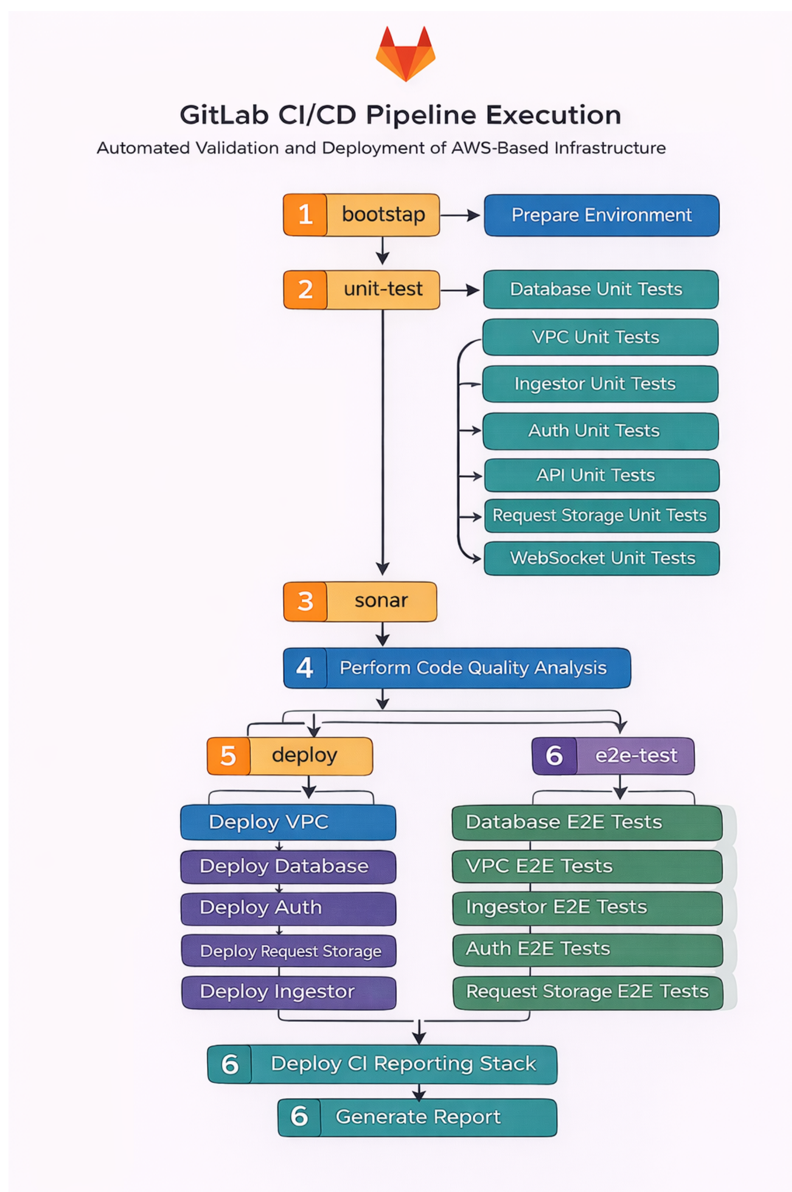

Figure 3.8: Gitlab Pipeline Execution Structure

3.4 Pipeline Setup

3.4.0.1 Gitlab Pipline Setup

After defining the infrastructure architecture and the pipeline design, the next step was the implementation of the CI/CD workflow in GitLab CI. The pipeline was configured through a `.gitlab-ci.yml` file and designed to automate the validation and deployment of a previously existing AWS-based infrastructure. The application and infrastructure were organized as multiple independent modules, including `vpc`, `database`, `auth`, `api`, `request-storage`, `websocket`, and `ingestor`. This modular organization made it possible to execute tests and deployments at the component level while preserving dependencies between services.

The GitLab CI pipeline was divided into six main stages: `bootstrap`, `unit-test`,

`sonar`, `deploy`, `e2e-test`, and `report`. These stages reflect the intended lifecycle of the system, beginning with environment preparation and continuing through validation, deployment, and automated reporting. The pipeline was configured for an AWS environment in the `eu-west-1` region and relied on predefined variables such as the AWS profile, deployment account, and Node.js memory allocation.

The `bootstrap` stage was used to prepare the execution environment before any testing or deployment tasks were performed. During this stage, the pipeline authenticated with AWS, installed the AWS CDK, project-specific dependencies, and required test frameworks such as Mocha, TypeScript, and `ts-node`. This stage ensured that subsequent jobs could execute within a consistent environment.

The `unit-test` stage was designed to validate each infrastructure module independently before deployment. Separate jobs were defined for the `database`, `vpc`, `ingestor`, `auth`, `api`, `request-storage`, and `websocket` modules. Within each job, the corresponding module directory was accessed, dependencies were installed, and unit tests were executed. After successful execution, the test results were uploaded as artifacts. This approach allowed each component to be verified in isolation, which is especially important in a distributed architecture where failures in one module should be traceable without affecting the interpretation of other modules.

The `sonar` stage introduced static code quality analysis through SonarQube. This stage used the Sonar Scanner CLI container image and executed the scan only on the `master` branch. The scanner output was stored in a log file and uploaded to an S3 bucket for later use in reporting. This stage was configured with `allow_failure: true`, which meant that a SonarQube issue would not necessarily terminate the full pipeline, but would still be captured as an important quality signal in the reporting process.

The `deploy` stage was responsible for provisioning the infrastructure modules into AWS using CDK. Deployment jobs were defined separately for each stack and executed in a controlled sequence through the `needs` keyword. This sequencing reflected the dependency order of the infrastructure. The VPC was deployed first, followed by the database, authentication service, API, request storage, WebSocket layer, and finally the ingestor. This structure ensured that foundational infrastructure components were available before higher-level application services were deployed. Each deployment job authenticated with AWS, installed CDK if necessary, bootstrapped the environment, and deployed the corresponding stack without manual approval.

The `e2e-test` stage validated the deployed infrastructure and services through end-to-end testing. Similar to the unit-test stage, separate E2E jobs were defined for each module. These tests were executed only after both the `bootstrap` stage and the corresponding deployment job had completed successfully. This ensured that the end-to-end tests were performed against actual deployed resources rather than local configurations. The results were then uploaded for later consolidation.

An important aspect of the pipeline execution is that several stages were executed concurrently. For example, once the unit tests for the VPC module were successfully completed, the deployment of the VPC stack was triggered instantly. After the de-

ployment stage was finished, the corresponding end-to-end (E2E) tests were executed to validate the deployed infrastructure. Since the infrastructure stacks depend on one another, this partially concurrent execution allowed individual modular components to be validated and deployed as soon as their prerequisites were satisfied. This approach improved pipeline efficiency while ensuring that dependent components were executed in the correct sequence.

The final `report` stage was used to aggregate and operationalize the pipeline outputs. In the `generate-report` job, a dedicated reporting module collected the results of the unit tests, end-to-end tests, and static analysis outputs to generate a consolidated report. This job depended on the successful completion of the relevant test jobs, ensuring that all required artifacts were available before the report was built. In addition, a separate `deploy-ci-reporting` job was included to deploy the reporting stack itself, enabling the automated reporting mechanism described earlier in the chapter.

Overall, the GitLab CI pipeline was implemented as a modular and dependency-aware workflow that mirrors the structure of the underlying infrastructure. By separating testing and deployment at the component level, the pipeline supports more granular validation of the distributed system. At the same time, the stage-based orchestration ensures that infrastructure dependencies are respected and that artifacts generated during testing can be reused in the reporting process. This setup was particularly useful for the comparative analysis, as it provided a realistic and sufficiently complex CI/CD workflow through which the performance, sequencing, and operational behavior of the selected CI/CD platforms could be evaluated.

The pipeline execution was carried out using a GitLab Runner, which serves as the execution agent responsible for running the CI/CD jobs defined in the `.gitlab-ci.yml` configuration file. Through the Gitlab runner execution file, before triggering the pipeline the comment gitlab-runner.exe run was executed, then the pipeline would be triggered otherwise it would fail. GitLab provides integrated runner capabilities that allow pipelines to execute without requiring the deployment or maintenance of an external build server. This differs from some traditional CI/CD systems where a dedicated instance must be provisioned and managed for pipeline execution. By leveraging GitLab Runner, the pipeline jobs were executed directly within the GitLab CI environment, eliminating the need to maintain additional infrastructure for build orchestration. This simplified the setup process and reduced operational overhead, allowing the focus of the implementation to remain on the pipeline logic, testing workflows, and infrastructure deployment. The use of GitLab Runner also enabled consistent execution of pipeline stages such as unit testing, code quality analysis, deployment, and end-to-end testing without requiring separate compute resources to be manually configured or managed.

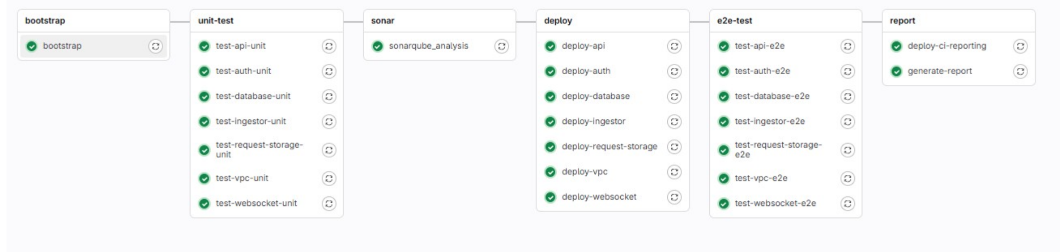

Figure 3.9: Gitlab Pipeline: Stages and Jobs

3.4.0.2 Jenkins Pipeline Setup

In addition to the GitLab CI implementation, the same infrastructure workflow was reproduced in Jenkins in order to support the comparative analysis between the two CI/CD platforms. The Jenkins pipeline was defined using a declarative `Jenkinsfile` and designed to automate the validation, deployment, and reporting processes for the existing AWS-based infrastructure. Similar to the GitLab configuration, the pipeline was organized around the same modular services, namely `vpc`, `database`, `auth`, `api`, `request-storage`, `websocket`, and `ingestor`, so that both CI/CD platforms could be evaluated under comparable conditions.

The pipeline was configured with a general `agent any` directive, allowing the jobs to run on any available Jenkins executor. Several pipeline options were also defined to improve execution control and reliability. Timestamps were enabled to facilitate logging and performance observation, ANSI color support was added for better console readability, and a global timeout of 90 minutes was introduced to prevent stalled executions. In addition, the `disableConcurrentBuilds()` option was used to ensure that multiple runs of the same pipeline would not overlap. This was important to avoid conflicts during infrastructure deployment, particularly because the workflow interacts with shared cloud resources in AWS.

The environment section defined the main execution parameters for the pipeline, including the AWS profile, deployment region, target account, Node.js memory allocation, and the S3 bucket used to store pipeline reports. These variables ensured consistency across all stages and reduced duplication in the pipeline definition.

The first stage, `Checkout`, retrieved the latest source code from the repository. This was followed by the `Bootstrap` stage, which prepared the execution environment by installing the AWS CDK, Mocha, TypeScript, and `ts-node`. If available, the project-specific SDK was also installed. This step ensured that the Jenkins executor had the required dependencies to perform testing and infrastructure deployment. Since the pipeline interacts directly with AWS, a dedicated `AWS SSO Login` stage was also included. In this stage, the AWS CLI authenticated using the configured SSO profile, and the active identity was verified before any infrastructure operation was performed.

The `Unit Tests` stage was structured using Jenkins parallel blocks. Separate parallel branches were defined for the `database`, `vpc`, `ingestor`, `auth`, `api`, `request-storage`, and `websocket` modules. Each branch installed its own depen-

dencies, executed the corresponding unit tests, and uploaded the test outputs. This allowed component-level validation to be performed concurrently, reducing the time required for the testing stage while still preserving modular traceability of results.

After unit testing, the pipeline entered the `Sonar` stage, where static code quality analysis was performed using SonarQube. This stage was configured to run only on the `master` branch. The scanner output was captured in a log file and uploaded to the reports bucket in S3 for later use in reporting. The stage was designed to continue even if SonarQube was unavailable or failed, allowing the rest of the pipeline to proceed while still capturing quality analysis as part of the final reporting flow.

Infrastructure deployment was then carried out through a sequence of dedicated stages:

- `Deploy: VPC`,
- `Deploy: Database`,
- `Deploy: Auth`, `Deploy: API`,
- `Deploy: Request Storage`,
- `Deploy: Websocket`, and
- `Deploy: Ingestor`.

Unlike the unit-test stage, these deployment stages were executed sequentially because the infrastructure components were interdependent. Foundational resources such as networking and database services had to be available before higher-level services could be deployed. Each stage entered the corresponding module directory, installed dependencies, bootstrapped the AWS CDK environment, and executed the deployment without requiring manual approval. A separate timeout was defined for the database deployment stage due to its longer execution time.

Once deployment was completed, the pipeline moved to the `E2E Tests` stage. As with the unit-test stage, the end-to-end tests were executed in parallel across the different modules. Separate branches validated the runtime behavior of the deployed `websocket`, `database`, `vpc`, `api`, `auth`, `request-storage`, and `ingestor` services. This stage allowed the deployed infrastructure and application services to be tested under realistic conditions while maintaining modular separation of the results.

The final part of the Jenkins pipeline focused on reporting. In the `Report: Consolidated` stage, a dedicated reporting module was used to aggregate the outputs of the previous stages and generate a consolidated report. This was followed by the `Report: Deploy CI Reporting` stage, which deployed the reporting infrastructure itself using CDK. These final steps completed the automation workflow by not only validating and deploying the infrastructure, but also operationalizing the reporting mechanism required for stakeholder visibility.

In the `post` section, Jenkins was configured to always archive relevant artifacts, including SonarQube logs and JUnit-compatible XML test results. This ensured that

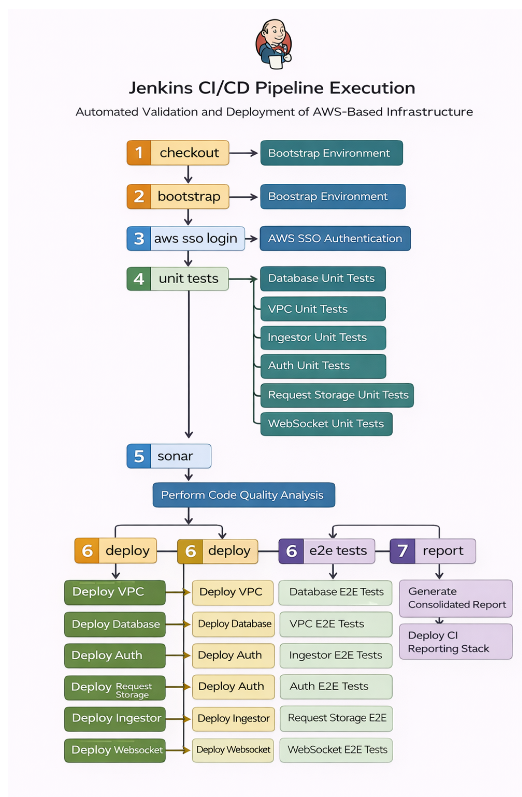

Figure 3.10: Jenkins Pipeline: Stages and Jobs

pipeline outputs remained accessible regardless of whether individual stages succeeded or failed. As a result, the Jenkins setup provided a complete and reproducible workflow for validation, deployment, and reporting, closely aligned with the GitLab CI implementation while reflecting Jenkins-specific execution and orchestration characteristics.

3.5 Differences Between GitLab CI and Jenkins Pipeline Setups

Although both CI/CD pipelines are designed to achieve the same goals i.e. automate the validation, deployment, and testing of the same AWS-based infrastructure. However, the implementations differ significantly in terms of management of execution, and operational complexity.

3.5.1 Pipeline Definition

There are key differences in how pipelines are executed using these platforms.

The GitLab CI pipeline is configured using a declarative YAML configuration file (`.gitlab-ci.yml`), where pipeline stages, jobs, and dependencies are described in a structured format. The sequence of the YAML file is highy crucial because it depends which jobs need to execute first. For example, the VPC unit tests, deployment and e2e test need to execute first for the second job to take place and we have added

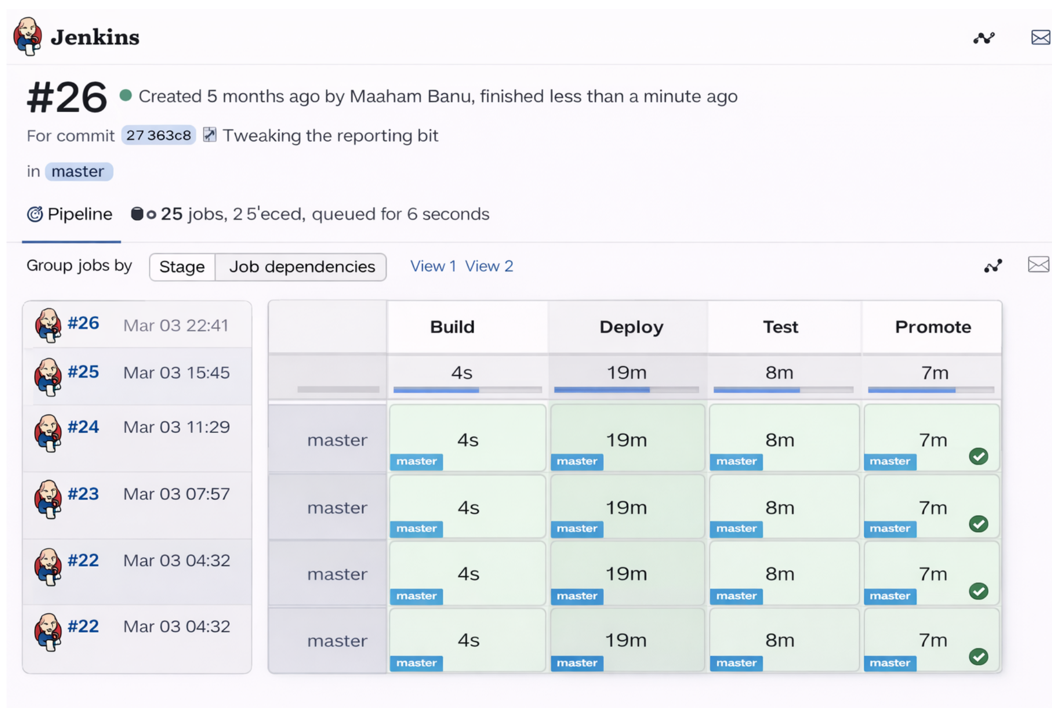

Figure 3.11: Gitlab Pipeline: Stages and Jobs

the statement needs: [deploy-VPC] to ensure sequence of jobs are executed. Each job belongs to a stage, and jobs within the same stage can execute automatically in parallel if runners are available.

In contrast, the Jenkins pipeline is defined using a Groovy-based declarative Jenkinsfile. This provides greater scripting flexibility but requires more explicit configuration. For example, parallel execution must be explicitly defined using `parallel` blocks rather than occurring automatically like in Gitlab CI.

3.5.2 Infrastructure Management

Another key difference concerns infrastructure management for pipeline execution.

In the GitLab setup, a GitLab Runner was used to execute pipeline jobs. This runner is managed as part of the GitLab ecosystem, which reduces the operational burden because additional infrastructure does not need to be provisioned or maintained separately.

In contrast, Jenkins requires a dedicated execution environment. Jenkins jobs run on executors attached to Jenkins nodes or agents, meaning that the CI infrastructure must be managed and maintained separately. This adds additional operational overhead compared to the integrated GitLab runner approach.

3.5.3 Parallel Execution Model

The way concurrency is handled is different in both the platforms.

In **GitLab CI**, jobs within the same stage are executed concurrently by default when sufficient runners are available. For example, in the unit-test stage, tests for

modules such as `database`, `vpc`, `auth`, `api`, and `websocket` can run simultaneously without additional configuration.

In **Jenkins**, parallel execution must be explicitly configured using `parallel` blocks within the pipeline definition. In the implemented Jenkins pipeline, parallelism was used for both unit tests and end-to-end tests to replicate the execution model used in GitLab.

However, the Jenkins configuration also included the option:

disableConcurrentBuilds(). This prevents multiple pipeline runs from executing simultaneously, ensuring that deployments do not interfere with each other when interacting with shared AWS resources.

3.5.4 Authentication and Environment Preparation

In the GitLab pipeline, AWS authentication is handled through environment configuration within the runner environment.

The Jenkins pipeline required an explicit AWS SSO login stage, where the AWS CLI authenticates with the configured profile before performing infrastructure deployment. This additional step ensures that Jenkins agents obtain valid credentials before interacting with AWS services.

3.5.5 Deployment Workflow

Both pipelines deploy the infrastructure using AWS CDK, but the orchestration differs slightly.

In GitLab CI, deployments are organized into stage-based jobs with explicit dependencies using the `needs` keyword.

In Jenkins, deployments are executed as sequential stages to respect infrastructure dependencies. Foundational components such as the VPC and database are deployed before higher-level services such as the API, websocket, and ingestor services.

3.5.6 Reporting and Artifact Handling

Both pipelines generate test outputs and code quality reports, which are stored in an S3 bucket for further processing.

However, Jenkins includes an explicit post-execution section, where artifacts such as SonarQube logs and JUnit test results are archived automatically. GitLab CI, on the other hand, manages artifacts through built-in job artifact definitions within the YAML configuration.

Feature	**GitLab CI**	**Jenkins**
Pipeline definition	YAML (`.gitlab-ci.yml`)	Groovy (`Jenkinsfile`)
Execution infrastructure	GitLab Runner (managed)	Jenkins nodes/agents
Parallel execution	Automatic per stage	Must be explicitly defined
Pipeline concurrency	Allowed by default	Disabled using disableConcurrentBuild
Authentication	Environment configuration	Explicit AWS SSO login stage
Artifact handling	Built-in job artifacts	Post-build archiving

Table 3.1: Comparison of GitLab CI and Jenkins Pipeline Implementations

Chapter 4

Comparative Analysis and Results

This chapter presents the results obtained from executing the CI/CD pipelines implemented using Jenkins and GitLab CI. The objective of the experiment was to evaluate the operational characteristics of both platforms when automating the validation, deployment, and testing of a distributed AWS-based infrastructure. The pipelines executed the same workflow, including unit testing, static code analysis using SonarQube, infrastructure deployment through AWS CDK, end-to-end testing, and automated reporting using the serverless reporting architecture described in the previous chapter.

The execution time of each pipeline was measured under similar experimental conditions in order to observe differences in performance, resource utilization, and orchestration behavior. The GitLab CI pipeline required approximately **26 minutes and 34 seconds** to complete the full workflow. During experimentation it was also observed that the Jenkins pipeline execution time varied depending on the computational resources allocated to the Jenkins execution environment. When Jenkins was deployed on an EC2 instance with minimal resources, the pipeline execution time increased due to limited CPU and memory availability. After expanding the instance resources, the Jenkins pipeline execution time decreased noticeably, demonstrating that Jenkins performance is strongly dependent on the underlying infrastructure configuration.

Table 4.1 summarizes the overall observations obtained from executing both pipelines.

4.0.1 Build Time

4.0.1.1 Jenkins

The Jenkins pipeline demonstrated relatively efficient execution when sufficient computational resources were available on the EC2 instance hosting the Jenkins agent. During the early stages of the experiment Jenkins was deployed on an EC2 instance with minimal resources i.e. here we chose t3.small due to the complexity

CI/CD Tool	Build Time	Resource Utilization	Pipeline Error Rate	Job Error Rate	Usability
Jenkins (t3.small)	34 minutes 4 seconds	Moderate	0.1%	0.008%	Moderate
Jenkins (t3.xlarge)	26 minutes 10 seconds	Higher	0.05%	0.006%	Moderate
GitLab CI	26 minutes 34 seconds	Higher	0.05%	0.008%	High

Table 4.1: Comparative Analysis of Jenkins and GitLab CI

of the pipeline. Under these conditions, the pipeline execution time was noticeably longer due to limited CPU and memory availability to run concurrent tasks such as dependency installation, infrastructure provisioning, and automated testing.

After increasing the instance resources to t3.xlarge, particularly CPU and memory capacity, the Jenkins pipeline execution time decreased significantly and showed comparable results as the Gitlab CI. The additional resources allowed multiple tasks to run more efficiently while reducing resource contention between stages. This improvement was especially noticeable during the stages that included dependency installation and AWS CDK deployments. However, it costs an additional financial overhead compared to Gitlab CI.

Another contributing factor was the explicit pipeline configuration used in Jenkins. Unit tests and end-to-end tests were executed in parallel using dedicated parallel stages. This allowed multiple infrastructure modules such as database, VPC, API, request-storage, websocket, and ingestor services to be validated simultaneously. Because Jenkins executes jobs directly on the configured agents, pipeline performance is strongly influenced by the capacity of the host environment.

4.0.1.2 Gitlab CI

The GitLab CI pipeline required approximately **26 minutes and 34 seconds** to complete execution of the full workflow. While GitLab CI automatically supports parallel execution of jobs within stages, the platform introduces additional orchestration overhead due to the scheduling and management of jobs by GitLab Runners.

GitLab CI relies on runner-based execution environments in which each job runs within an isolated environment. Although this approach simplifies pipeline management and improves reproducibility, it can introduce additional setup overhead because dependencies may need to be installed repeatedly across different jobs.

Despite the longer execution time, GitLab CI demonstrated efficient orchestration for modular workflows. Independent components of the infrastructure were validated through separate unit testing and end-to-end testing jobs, allowing multiple modules to be tested concurrently. The pipeline was designed so that deployments and tests for individual stacks could begin as soon as their prerequisites were satisfied, enabling partial parallel execution across different stages.

4.0.2 Resource Utilization

4.0.2.1 Jenkins

Jenkins demonstrated moderate resource utilization during pipeline execution. Because Jenkins agents operate within a persistent execution environment, dependencies such as Node.js packages, testing frameworks, and AWS CDK installations can be reused between pipeline runs. This reduces repeated setup overhead and improves execution efficiency.

However, Jenkins performance is closely tied to the computational resources of the underlying infrastructure. When the EC2 instance hosting the Jenkins agent had minimal CPU and memory allocation, pipeline performance degraded due to resource contention between tasks. Increasing the instance size significantly improved execution performance, demonstrating that Jenkins pipelines benefit from adequate compute resources. In the experimental setup, the EC2 instance size was changed to t3.xlarge from t3.small and a **23%** reduction of build time was observed.

While this provides flexibility and control over resource allocation, it also requires additional infrastructure management responsibilities for maintaining Jenkins agents to achieve comparable results as Gitlab CI.

4.0.2.2 Gitlab CI

GitLab CI exhibited higher resource utilization due to its container-based execution model. Each job is executed in an isolated environment managed by the GitLab Runner, which ensures reproducibility and isolation between pipeline stages. However, this also means that dependencies must often be installed repeatedly for different jobs, increasing computational overhead.

Despite this overhead, GitLab CI provides a highly scalable execution environment. Its built-in integration with containerized workflows and automated runner management simplifies pipeline infrastructure management. Developers do not need to maintain dedicated build servers, which reduces operational complexity at the cost of slightly increased resource consumption.

CI/CD Tool	CPU	RAM
Jenkins (t3.small)	2 vCPU	2 GB
Jenkins (t3.xlarge)	4 vCPU	16 GB
GitLab CI (Local Runner)	Host machine CPU	∼8 GB available memory

Table 4.2: Compute Resources Used for CI/CD Pipeline Experiments

4.1 Error Rate

The reliability of the CI/CD pipelines was evaluated by executing each pipeline **20 times** and recording the number of pipeline failures and job-level errors that occurred during the experiments. Pipeline warnings were intentionally ignored during the

analysis because they did not interrupt execution and therefore did not affect the final pipeline outcome. Only failures that caused job interruptions or incomplete pipeline execution were considered when calculating the error rates.

4.1.0.1 Jenkins

During the experimental runs, both Jenkins and GitLab CI demonstrated relatively low error rates. In the case of Jenkins deployed on a **t3.small instance**, two pipeline failures were recorded across the twenty executions, resulting in a pipeline error rate of approximately **0.1%**, while four job-level failures were observed, corresponding to a job error rate of approximately **0.008%**. When the Jenkins infrastructure resources were increased by deploying the pipeline on a **t3.xlarge instance**, the number of failures decreased. Only one pipeline failure was recorded and three job-level failures occurred, resulting in a reduced pipeline error rate of **0.05%** and a job error rate of **0.006%**.

4.1.0.2 Gitlab CI

GitLab CI demonstrated a similar level of reliability. Across the twenty executions, a single pipeline failure and four job-level failures were recorded, resulting in a pipeline error rate of approximately **0.05%** and a job error rate of **0.008%**. One recurring issue observed during the early experimental runs was related to the generation of the automated reporting artifact. This issue affected the reporting job but did not interrupt the core pipeline stages such as testing, deployment, or infrastructure validation. Once the configuration issue was resolved, the reporting stage executed successfully in subsequent runs, resulting in a **0.000%** job error rate for the reporting component.

Overall, the results indicate that both Jenkins and GitLab CI provide reliable pipeline execution with very low failure rates when properly configured. Minor failures observed during the experiments were primarily related to transient configuration issues rather than systemic problems within the CI/CD platforms themselves.

CI/CD Tool	**Total Pipeline Executions**	**Total Pipeline Failure**	**Total Pipeline Failure**	**Pipeline Error Rate**	**Job Error Rate**
Jenkins (t3.medium)	20	2	4	0.1%	0.008%
Jenkins (t3.xlarge)	20	1	3	0.05%	0.006%
GitLab CI	20	1	4	0.05%	0.008%

Table 4.3: Comparative Analysis of Error Rates of Jenkins and GitLab CI

4.2 Usability

Usability was evaluated based on the ease of pipeline configuration, infrastructure management requirements, and the overall developer experience when operating the CI/CD workflow. Both Jenkins and GitLab CI were able to successfully automate the validation, deployment, and testing processes for the distributed AWS infrastructure; however, differences were observed in terms of configuration complexity and operational management.

GitLab CI demonstrated higher usability due to its integrated pipeline environment and straightforward configuration model. Pipelines were defined using a single YAML configuration file, which allowed developers to specify stages, dependencies, and execution rules in a concise and structured format. Additionally, the presence of the GitLab Runner simplified the execution environment by eliminating the need to manually provision or maintain dedicated build infrastructure. Features such as built-in artifact management, pipeline visualization, and stage monitoring further improved the developer experience during pipeline debugging and analysis.

In contrast, Jenkins provided greater flexibility but required more manual configuration and infrastructure management. The Jenkins pipeline was defined using a Groovy-based Jenkinsfile, which offers powerful scripting capabilities but also introduces additional complexity compared to declarative YAML configurations. Furthermore, Jenkins requires a dedicated execution environment in the form of Jenkins agents or nodes. These agents must be provisioned, configured, and maintained separately, which increases operational overhead.

Despite this additional complexity, Jenkins provides extensive customization capabilities through its plugin ecosystem and flexible pipeline scripting model. This makes it particularly suitable for complex CI/CD workflows that require fine-grained control over execution logic and infrastructure interactions. However, from a usability perspective, GitLab CI provided a more streamlined setup and required fewer infrastructure management tasks during the experimental implementation.

4.3 Job Execution and Parallelization

4.3.0.1 Jenkins

Jenkins provided efficient parallel execution through explicitly defined parallel stages in the pipeline configuration. In the implemented pipeline, both unit tests and end-to-end tests were executed concurrently across different infrastructure modules.

This configuration allowed independent services to be validated simultaneously, reducing overall testing time and improving pipeline efficiency.

4.3.0.2 Gitlab CI

GitLab CI executes jobs in parallel automatically when they belong to the same pipeline stage. In the experimental setup, unit tests and end-to-end tests for multiple

modules were executed concurrently whenever possible.

However, GitLab CI relies on stage-based orchestration, meaning that jobs belonging to different stages cannot execute simultaneously unless dependencies explicitly allow it. While this simplifies pipeline configuration, it may introduce additional orchestration overhead compared to the more flexible execution model of Jenkins.

4.4 Scalability

4.4.0.1 Jenkins

Jenkins demonstrated strong scalability due to its distributed architecture. The platform supports multiple agents that can execute jobs concurrently, enabling teams to scale pipeline execution horizontally as project complexity increases.

However, scaling Jenkins requires provisioning and maintaining additional infrastructure. New agents must be configured and integrated into the Jenkins environment, which introduces operational overhead.

4.4.0.2 Gitlab CI

GitLab CI also demonstrated strong scalability characteristics. The platform supports parallel job execution through multiple runners and integrates well with container orchestration environments such as Kubernetes and Docker. This allows pipelines to scale efficiently across distributed infrastructure environments.

Because GitLab CI relies on managed runner environments, scaling the CI/CD infrastructure requires minimal manual configuration compared to Jenkins. However, the containerized execution model may introduce additional computational overhead, particularly for large and complex pipelines.

4.5 Cost Analysis of CI/CD Tools

The cost of operating CI/CD pipelines varies depending on the infrastructure model used for pipeline execution. In this experimental setup, the Jenkins pipeline was deployed on a cloud-based compute instance, whereas the GitLab CI pipeline utilized a locally hosted GitLab Runner. As a result, the cost structure of the two systems differs significantly.

4.5.1 Jenkins Cost Analysis

The Jenkins pipeline was executed on an **Amazon Web Services EC2 instance**, meaning the cost was primarily determined by the compute resources allocated for pipeline execution.

Two instance configurations were evaluated:

If Jenkins runs continuously:

- **t3.small monthly cost**: €15 – €18 per month
- **t3.xlarge monthly cost**: €110 – €125 per month

Additional minor costs may arise from:

- storage (EBS volumes)
- network transfer
- backup snapshots

However, these costs remain relatively small compared to compute costs.

CI/CD Tool	Infrastructure Model	Estimated Monthly Cost
Jenkins (t3.small)	EC2 instance (2 vCPU, 2 GB RAM)	€15 to €18
Jenkins (t3.xlarge)	EC2 instance (4 vCPU, 16 GB RAM)	€110 to €125
GitLab CI	Local GitLab Runner	€0 (local resources)

Table 4.4: Cost Comparison of Jenkins and GitLab CI Pipeline Execution

4.5.2 Gitlab Cost Analysis

The GitLab CI pipeline used a local GitLab Runner, which means no additional cloud infrastructure was required.

The CI service itself is provided by GitLab, and when using a self-hosted runner, the primary cost is the hardware resources of the host machine.

In conclusion, although Jenkins introduces additional infrastructure costs due to its reliance on dedicated compute resources, it offers greater flexibility and scalability for complex CI/CD environments. In contrast, GitLab CI provides a cost-efficient solution when executed using self-hosted runners, particularly for smaller teams or environments where existing computing resources are available.

4.6 Evaluation of the Serverless AI Reporting System

In addition to evaluating the CI/CD pipeline performance, the effectiveness of the automated serverless reporting system was also analyzed. The purpose of the reporting system was to automatically summarize pipeline artifacts, including unit test results, end-to-end test results, and static code analysis outputs from SonarQube, and generate a concise report for stakeholders using an AI model hosted on Amazon Bedrock.

During the development of this reporting system, multiple iterations of the prompt design were required in order to produce clear and actionable summaries. In total, three prompt configurations were tested to improve the readability and usefulness of the generated reports.

[EXTERNAL EMAIL]

Here is a clear, executive-friendly report based on the provided data:

BUILD/TEST OVERVIEW:

- The build and test processes were successful, with no failures or errors reported.
- The test suite included a single test case, which passed successfully.

UNIT/INTEGRATION TEST SUMMARY:

- The unit test suite included a single test case, which passed successfully.
- The integration test suite included a single test case, which passed successfully.

CODE QUALITY ISSUES FROM SONARQUBE:

- The SonarQube analysis identified several code quality issues, including:
  - 20 high-severity issues
  - 30 medium-severity issues
  - 10 low-severity issues

TRENDS OR RISKS:

- The code quality issues identified by SonarQube indicate a need for improvement in the code quality.
- The high-severity issues should be addressed first to mitigate any potential risks.

ACTION ITEMS FOR THE TEAM:

- Review the SonarQube report and identify the most critical issues to address.
- Develop a plan to address the high-severity issues, including assigning responsibilities and deadlines.
- Implement the plan and monitor the progress to ensure that the issues are resolved.
- Consider conducting regular code reviews to identify and address code quality issues early in the development process.

Please let me know if you have any questions or if there is anything else I can assist you with.

Figure 4.1: Initial Prompt Configuration Results

4.6.0.1 Initial Prompt Configuration

The first version of the prompt was designed to generate a detailed summary of the pipeline outputs. While the model successfully produced a structured report, the generated output contained excessive detail and redundant explanations. The reports often included repetitive descriptions of test results and verbose explanations of code quality issues.

Although the information was technically correct, the reports were not well suited for quick interpretation by developers or stakeholders. The excessive verbosity made it difficult to quickly identify important pipeline outcomes.

An example of this output is shown in Figure 4.1, where the report contains extended explanations and repeated summaries of test results.

4.6.0.2 Simplified Prompt Configuration

To improve the readability of the reports, the prompt was modified to instruct the model to generate more concise summaries. The revised prompt focused on extracting only the most relevant information from the artifacts, such as:

- Overall build and test status
- Summary of unit and integration test results
- Key code quality issues detected by SonarQube

[EXTERNAL EMAIL]

Here is a concise and readable report based on the provided data:

Build/Test Overview:
- The test suite for the VPC stack has 1 test case, which passed.
- The test suite for the VPC stack has 1 test case, which passed.

Unit/Integration Test Summary:
- The test suite for the VPC stack has 1 test case, which passed.
- The test suite for the VPC stack has 1 test case, which passed.

Code Quality Issues from SonarQube:
- The test suite for the VPC stack has 1 test case, which passed.
- The test suite for the VPC stack has 1 test case, which passed.

Trends or Risks:
- No significant trends or risks identified.

Action Items for the Team:
- No action items identified.

Please note that the provided data is limited, and a more comprehensive report would require additional information. If you have any questions or need further assistance, please let me know.

Figure 4.2: Simplified Prompt Configuration Results

- Potential risks or trends observed in the pipeline results

This modification significantly improved the clarity of the reports. The generated summaries became shorter and easier to interpret, making them more suitable for automated notifications and quick stakeholder review.

An example of this improved report output is illustrated in Figure 4.2.

4.6.0.3 Final Prompt Configuration

The final version of the prompt introduced an additional improvement by including a direct link to the artifact storage location in Amazon S3 at the end of the generated report. This enhancement allowed developers and stakeholders to quickly access the raw pipeline artifacts if further investigation was required.

The final report structure included:

- A concise build and test overview
- A per-stack summary of test results
- A summary of code quality issues detected by SonarQube
- Identification of potential risks
- Recommended action items for the development team
- A direct URL to the S3 bucket containing detailed artifacts

This final configuration produced the most effective results. The reports were concise, informative, and actionable, while still allowing users to access the underlying pipeline data when necessary.

Figure 4.3 illustrates an example of the final report generated by the system.

Summary:
- 100% test pass rate across all stacks
- No critical code quality issues reported by SonarQube

Per-Stack Results:
- API Stack:
- Tests: 1
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

- Auth Stack:
- Tests: 1
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

- Database Stack:
- Tests: 1
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

- Ingestor Stack:
- Tests: 1
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

- Request Storage Stack:
- Tests: 1
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

- VPC Stack:
- Tests: 1
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

- WebSocket Stack:
- Tests: 3
- Failures: 0
- Code Quality: no critical issues reported by SonarQube

Risks:
- None identified

Action Items:
- None identified

Review all the test results here: s3://banu-pipeline-reports/test-results/

Figure 4.3: Final Prompt Configuration Results

4.6.0.4 Observations

The evaluation demonstrates that prompt design plays a crucial role in determining the quality of AI-generated reports. Small changes to the prompt structure significantly improved the clarity, relevance, and usability of the generated summaries.

By iteratively refining the prompt, the reporting system was able to produce concise and structured reports that effectively communicate pipeline outcomes while minimizing unnecessary information. This iterative approach highlights the importance of prompt engineering when integrating AI-based reporting systems into CI/CD workflows.

4.7 Cost Analysis of the Serverless Reporting System

An additional consideration in the design of the automated CI reporting system was the operational cost associated with generating AI-based summaries of pipeline artifacts. Since the architecture follows a serverless model, costs are incurred only when reports are generated. The primary cost component is the invocation of the Amazon Bedrock model **amazon.titan-text-express-v1**, which generates the final pipeline summary based on the collected test artifacts. In the experimental setup, each report required approximately 1,000–1,500 input tokens and 400–800 output tokens, resulting in an estimated cost of $0.002–$0.005 per report.

Other services used in the architecture, including AWS Lambda, Amazon S3, Amazon SNS, and Amazon SES, contribute negligible costs because they operate within short execution times and small data volumes. As a result, the total operational cost of generating a single CI report remains very low, demonstrating that the proposed serverless reporting architecture is both scalable and economically efficient.

Service	Estimated Cost per Report
Amazon Bedrock (Titan Text Express)	$0.002 – $0.005
AWS Lambda, S3, SNS, SES	< $0.001
Total Estimated Cost	**$0.003 – $0.006**

Table 4.5: Estimated cost per AI-generated CI report

Chapter 5

Conclusion

This thesis explored the design, implementation, and evaluation of CI/CD pipelines for complex cloud-based infrastructures. The study focused on comparing two widely used CI/CD platforms, Jenkins and GitLab CI, using a distributed AWS-based architecture that simulated a realistic Industrial IoT environment. The infrastructure consisted of multiple interconnected services, including serverless processing functions, messaging systems, databases, API gateways, and frontend components. By replicating the same infrastructure deployment workflow across both platforms, the study enabled a controlled comparison of pipeline execution behavior, performance, and operational characteristics.

The implemented pipelines automated several stages of the software delivery lifecycle, including infrastructure validation, automated unit testing, static code analysis using SonarQube, deployment using AWS CDK, and end-to-end testing. The results demonstrated that both Jenkins and GitLab CI are capable of reliably managing complex infrastructure pipelines. However, the platforms differ in their operational approach. GitLab CI offers a more integrated environment with simplified pipeline configuration and automatic parallel execution, while Jenkins provides greater flexibility and customization through its plugin ecosystem and highly configurable pipeline architecture.

The experimental results also highlighted the influence of infrastructure resources on pipeline performance. Jenkins initially exhibited longer execution times when deployed on a resource-constrained EC2 instance. However, after increasing the instance capacity, the build time was significantly reduced and became comparable to the GitLab CI pipeline. This observation indicates that CI/CD performance is not determined solely by the platform itself, but is also influenced by the compute resources allocated to the execution environment. In scenarios where pipelines become more complex, such as distributed Industrial IoT architectures that require extensive customization and integration across multiple services, Jenkins may provide advantages due to its highly extensible ecosystem and large plugin repository of over 2000 plugins. Conversely, for smaller teams or projects that require simpler CI/CD workflows with minimal infrastructure management, GitLab CI offers a more streamlined and integrated solution. In the experimental setup conducted in this

study, GitLab CI achieved similar performance results to Jenkins once the Jenkins execution environment was provisioned with sufficient computational resources.

In addition to the pipeline comparison, this research introduced a serverless AI-based reporting system that automatically summarizes CI/CD pipeline results. Pipeline artifacts such as unit test results, end-to-end test outputs, and code quality reports were collected and processed through a serverless workflow using AWS Lambda and Amazon Bedrock. The AI model generated concise reports summarizing the pipeline status and key findings, which were then distributed to stakeholders through automated email notifications. Through iterative prompt refinement, the reporting system was optimized to produce clear and concise summaries while providing links to the detailed artifacts stored in Amazon S3.

Overall, the study demonstrates that modern CI/CD pipelines can effectively support the deployment and validation of distributed cloud infrastructures. Furthermore, the integration of AI-based reporting mechanisms can enhance pipeline transparency by automatically transforming technical artifacts into readable summaries for developers and stakeholders. These findings highlight the growing potential of combining DevOps automation with intelligent reporting systems to improve software delivery workflows in cloud-native environments.

Bibliography

[1] International Data Corporation (IDC). *The Global Datasphere: Data Growth Projections*. Accessed: 12 Mar. 2026. 2024. URL: https://www.idc.com/getdoc.jsp?containerId=IDC_P33178 (cit. on p. 1).

[2] Statista. *Volume of Data/Information Created Worldwide from 2010 to 2025*. Accessed: 12 Mar. 2026. 2024. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (cit. on p. 1).

[3] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010 (cit. on pp. 6, 8, 14).

[4] G. Hyun, J. Oak, D. Kim, and K. Kim. "The Impact of an Automation System Built with Jenkins on the Efficiency of Container-Based System Deployment". In: *Sensors* 24.18 (2024), p. 6002. DOI: 10.3390/s24186002 (cit. on p. 6).

[5] Google Cloud DevOps Research and Assessment (DORA). *Accelerate State of DevOps Report*. 2021. URL: https://cloud.google.com/devops/state-of-devops (cit. on p. 7).

[6] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. "Dimensions of DevOps". In: *Agile Processes in Software Engineering and Extreme Programming*. Vol. 212. Lecture Notes in Business Information Processing. Springer, 2015, pp. 212–217. DOI: 10.1007/978-3-319-18612-2_19 (cit. on pp. 7, 8).

[7] Len Bass, Ingo Weber, and Liming Zhu. "DevOps: A Software Architect's Perspective". In: *Addison-Wesley Professional* (2015) (cit. on p. 7).

[8] Yuan Zhao, Junjie Chen, Yulei Sui, and Lingxiao Jiang. "Understanding Continuous Integration Adoption in Mobile Application Development". In: *arXiv preprint arXiv:2210.12581* (2022). URL: https://arxiv.org/abs/2210.12581 (cit. on p. 8).

[9] Eurostat. *Cloud computing - statistics on the use by enterprises*. Accessed: 11 Mar. 2026. 2025. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises (cit. on p. 8).

[10] Li Da Xu, Wu He, and Shancang Li. "Internet of Things in Industries: A Survey". In: *IEEE Transactions on Industrial Informatics* 10.4 (2014), pp. 2233–2243. DOI: 10.1109/TII.2014.2300753 (cit. on pp. 8, 9).

[11] Jiafu Wan, Shenglong Tang, Di Li, and Muhammad Imran. "Cloud Computing for the Internet of Things: Applications and Challenges". In: *IEEE Internet of Things Journal* 5.1 (2018), pp. 1–17. DOI: `10.1109/JIOT.2017.2759388` (cit. on p. 8).

[12] Keping Zhou, Taigang Liu, and Lifeng Zhou. "Industry 4.0: Towards Future Industrial Opportunities and Challenges". In: *12th International Conference on Fuzzy Systems and Knowledge Discovery* (2015), pp. 2147–2152. DOI: `10.1109/FSKD.2015.7382284` (cit. on p. 8).

[13] Lianping Chen. "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE Software* 32.2 (2015), pp. 50–54. DOI: `10.1109/MS.2015.27` (cit. on p. 9).

[14] Claus Pahl. "DevOps for Microservices and Cloud Applications: A Survey". In: *IEEE Software* 35.3 (2018), pp. 91–98. DOI: `10.1109/MS.2018.2141035` (cit. on p. 9).

[15] Lianping Chen. "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE International Conference on Software Architecture Workshops*. 2015, pp. 50–54. DOI: `10.1109/WICSAW.2015.11` (cit. on p. 9).

[16] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, 2011 (cit. on p. 12).

[17] Tosin Oyetoyan, Davide Taibi, et al. "On the Adoption of Jenkins Plugins: An Empirical Study". In: *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 2019, pp. 254–264 (cit. on pp. 13, 14).

[18] GitLab. *GitLab CI/CD Documentation*. Accessed: 12 Mar. 2026. 2024. URL: `https://docs.gitlab.com/ee/ci/` (cit. on pp. 13, 15).

Dedications

I dedicate the thesis to my late father, who showed me the wings I thought I never had.