Aradhana Gajera

Comparative Analysis of Jenkins, GitLab CI, and GitHub Actions: Performance Evaluation in CI/CD Pipelines

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

20 February 2025

Preface

This thesis is titled Comparative Analysis of Jenkins, GitLab CI, and GitHub Actions: Performance Evaluation in CI/CD Pipelines. Is the culmination of research into the performance and efficiency of CI/CD tools. The study aims to provide insights that are valuable with metrics such as build times, resource usage and error rates helping developers and organizations make informed decisions.

My interest in the field of cloud computing and DevOps and software deployment strategies inspired this research. With hands-on experience in setting up CI/CD pipelines using different tools. I came to realize the need for a comprehensive, data-driven comparison of these tools. The work was designed to contribute to both academic discussions and best practices in the industry.

Espoo, 20 February 2025
Aradhana Gajera

Abstract

Author:	Aradhana Gajera
Title:	Comparative Analysis of Jenkins, GitLab CI, and GitHub Actions: Performance Evaluation in CI/CD Pipelines
Number of Pages:	34 pages + 2 appendices
Date:	20 February 2025
Degree:	Master of Engineering
Degree Programme:	Information Technology
Professional Major:	Networking and Services
Supervisors:	Ville Jääskeläinen

Continuous integration and continuous deployment (CI/CD) pipelines are essential parts of modern software engineering. They offer a systematic approach to build testing and deployment environments of applications along with error reduction as well as improved software quality with accelerated development cycles. There are not many studies that have been conducted on the evaluation of these CI/CD tools.

This study provides a comprehensive evaluation of the three widely used CI/CD tools: Jenkins, GitLab CI and Github Actions on different use case scenarios by examining quantitative metrics such as build times, resource utilization and error rates. The study also addresses practical consideration for tool selection making it easier for developers to select the tools according to their needs.

Keywords: CI/CD Pipelines, GitHub Actions, Jenkins, GitLab CI, Build Time Optimization, Continuous Integration, Continuous Deployment, Docker, AWS EC2, DevOps Performance, Automation in Software Development, Resource Utilization, Error Rate Analysis, Software Engineering

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

List of Abbreviations

1 Introduction 1

1.1 Need of Study 4
1.2 Objectives 5
1.3 Methodology 6
1.4 Scope of Study and Its Limitations 7
1.5 Organization of Thesis 7

2 Literature Review 8

2.1 Evolution of CI/CD Practices and Industry Adoption 8
2.2 Introduction to Performance Metrics in CI/CD 10
2.3 Existing CI/CD Tools 12
2.4 Comparison of Tools 14
2.5 Challenges in Comparative Analysis 16

3 Setting up Test Environment 18

3.1 Jenkins Setup 18
3.2 GitLab CI Setup 19
3.3 GitHub Action Setup 20

4 Results 26

4.1 Build Time 26
4.2 Resource Utilization 27
4.3 Error Rates 28
4.4 Scalability 28
4.5 Job Execution and Parallelization 29

5 Discussions and Conclusions 31

References 33

Appendix 1 Jenkins Deployment file

Appendix 2 Jenkins Execution Summary Report

List of Abbreviations

CI	Continuous Integration
CD	Continuous Deployment
CI/CD	Continuous Integration/Continuous Deployment
SDLC	Software development lifecycle.

1 Introduction

Continuous Integration (CI) and Continuous Deployment (CD) practices have transformed the status quo of software development and operations, becoming main components in the world of modern software engineering methodologies. The need for streamlined and automated process for managing the lifecycle of software development has grown at exponential rate in the current landscape of DevOps and Agile Methodologies for software development (Shahin, 2015). CI/CD pipelines enable organizations to accelerate the software development lifecycle with high-quality standards, increased automation and minimized human intervention for minimizing the risks associated with manual process. This capability has changed the way the software is built, tested and deployed and how organizations approach software delivery lifecycle.

Traditionally, software development is characterized by long cycles of development and slow and delays in integrating changes and a high risk of failures in integration. As the development teams and software systems are becoming increasingly complex the traditional workflows inefficiencies became apparent. The early 2000s saw the emergence of CI as a practice that promoted frequent integration changes into a shared repository leading to a faster development lifecycle accompanied by automated testing. Martin Fowlers formalization of CI principals was pivotal change on emphasizing practices such as maintaining single source repository and automating builds for ensuring that tests are run frequently. This approach and practices which provide multiple feedback to the developers along with significantly reducing the risk of defects and integration challenges (Flower, 2024).

As CI (Continuous Integration) approach started to mature it naturally extended into Continuous Delivery and Continuous Deployment. Continuous Delivery focuses on ensuring that code is always in a deployable state through rigorous automated testing and quality assurance, Continuous Deployment takes this one

step further by automating the deployment process to production (Zaid, 2023). These practices embody the essence of CI/CD where software is continuously developed, integrated tested and the deployment in small, incremental steps. Organizations adopt quickly to the changing requirements which deliver value to the end-users more frequently.

CI/CD pipelines adoption have been driven by their alignment with the DevOps principles, which emphasize collaboration between the development and operations teams and automating the process of manual deployment. By implementing CI/CD, organizations are able to break down task between teams and foster a culture of responsibility which is shared and simplify the flow of work. As CI/CD focuses on iterative increment which enables teams to detect issues and deficiency in the development process early on leading to a robust software development lifecycle.

The increase of CI/CD tools has played a major role in the adoption of these practices. Some of the widely adapted tool such as Jenkins, GitLab CI and GitHub Actions have emerged as leading solutions each of them offering unique features and capabilities made for catering to the different needs of development teams. One of the early in the landscape and widely adopted tool Jenkins is known for its extensive plugin ecosystem and its flexibility, made it a preferred choice for organizations with complex requirements. GitLab CI one of the other popular tools in the market provides a tightly integrated experience within the GitLab and also helps in streaming workflows for teams that use GitLab for version control. GitHub Actions is relatively new tool in the market and has also rapidly increased its popularity because of it seamless incorporation into GitHub repositories and a marketplace of reusable workflows.

Despite the prevalent adoption, CI/CD pipelines are not without their own set of challenges. Implementing and maintaining these pipelines require significant investment in time resource and expertise. Navigating the issues such as tool selection, infrastructure setup and the integration of various components into a cohesive workflow. The task of ensuring the Scalability, reliability and security of

CI/CD pipelines can be overwhelming and complex particularly for organizations with large and distributed development teams.

One of the important aspects in the CI/CD is its impact on key performance metrics such as build times, resource utilization and error rates. Efficient CI/CD pipelines minimize build times, enabling developers to receive rapid feedback and maintain productivity.

Effective resource utilization ensures that CI/CD pipelines do not impose excessive operational costs degradation of the system performance. Low error rates indicate robustness and reliability of the CI/CD pipeline, reducing the likelihood of disruptions in the software delivery process. Understanding these metrics is essential for organizations to optimize the CI/CD pipelines maximizing the benefits (Shahin et al., 2017).

This thesis performs a comparative analysis of the three widely-used CI/CD tools Jenkins, GitLab CI and GitHub Actions to evaluate the efficiency in the handling of CI/CD workflows for software projects. By examining quantitative metrics such as build times, resource utilization and error rates. The study is designed to provide actionable insights into the strengths and limitations of these tools.

The study also addresses practical consideration for tool selection, this study contributes to the broader understanding of the CI/CD practices and its role in the modern software engineering landscape. By exploring the tradeoffs and challenges associated with different CI/CD tools the research targets to inform the best practices for implementing and optimizing CI/CD pipelines. In addition, the research aims to highlight the areas of future research, particularly in the context of emerging technologies and evolving methodologies.

In conclusion, there is a fundamental shift in how software is developed, tested and deployed with significant improvement in terms of speed quality and collaboration. Yet, realizing these benefits require careful consideration of the tools and strategies used to implement CI/CD pipelines. This study seeks to

contribute by providing detailed comparative and quantitative analysis for Jenkins, GitLab CI and GitHub Actions offering valuable insights for researchers and practitioners alike.

1.1 Need of Study

The rapidly changing scenario of the software development practices demand tools and workflows which optimize efficiency, reliability and scalability. In the current SDLC (software development lifecycle) CI/CD pipelines has become essential component for faster and seamless delivery of modern DevOps methodologies ensuring seamless development cycles and faster delivery of the product. Among different CI/CD tools in the market, GitHub Actions has emerged as popular choice due to its seamless integration with GitHub repositories and widespread ecosystem of reusable workflows.

The increasing reliance of GitHub Actions for modern CI/CD requires a thorough examination of its performance in comparison to established and other popular CI/CD tools like Jenkins and GitLab CI.

Error rates and build features and deployment issues can directly influence a pipelines reliability. Understanding how errors are handled by GitHub Actions and how it performs under such conditions is essential for users and organizations seeking to optimize their development process.

Although GitHub Actions has been widely adapted by teams ranging from small to moderate because of its ease of use there is still a gap in empirical evaluation for organizations seeking to make informed decision regarding the selection of CI/CD tools.

This study is essential for organizations who are evaluating the suitability of GitHub Actions in their CI/CD pipelines. The findings will add to a robust understanding of the tools capabilities, assisting informed decision-making and similarly to foster efficient deployment of the software in increasingly complex environments.

1.2 Objectives

This study dives into the comparative and quantitative analysis of the widely used CI/CD tools like GitHub Actions, Jenkins along with GitLab CI by the deployment of Python Django application on Amazon Elastic Compute Cloud (AWS EC2) instances using the Docker container. The evaluation focuses on key performance metrics like build times, resource usages along with usability and pipeline efficiency.

Evaluating the CI/CD tool performance by analyzing the build times and resource utilization and overall pipeline efficiency of GitHub Actions Jenkins and GitLab CI additionally investigating the usability aspects of this tools including ease of configuration along with integrations with existing workflows. This thesis aims to provide detailed insights on strengths, tradeoffs and limitations with associated CI/CD tool. Developing practical recommendations for selecting the most appropriate CI/CD tools based on the needs of the project. Objective of this thesis is to perform a comparison report on the basis of real world testing and empirical data. It also aims to highlight scenarios where GitHub Actions, Jenkins or GitLab CI are the most effective. It also aims to offer actionable insights for the software developers and organizations to optimize their CI/CD pipelines. Equip students and practitioners with an understanding of modern CI/CD tools landscape to enhance their decision making abilities.

The thesis aims to support academic and professional discussion on the selection and efficiency of CI/CD tools. Especially this thesis focuses on finding answers to the following research questions.

1. What are the strengths and weakness of GitHub Actions, Jenkins, and GitLab CI in managing pipelines?

2. How does GitHub Actions compare Jenkins and GitLab CI in terms of build time and resource usage for Python Django applications deployed in Dockerized environments on AWS EC2 instances?

3. What insights can be derived from the performance analysis to aid developers and organizations in selecting the most suitable CI/CD tool?

1.3 Methodology

To achieve the objectives mentioned previously, an extensive review of existing literature is carried out and the gaps are studied. The suitable tools to be considered are selected. The CI/CD pipeline is built for streamlining the software development program. Github Action workflow is a critical input data. The obtained results will be studied, interpreted, compared with existing test results and discussed to reach a conclusion.

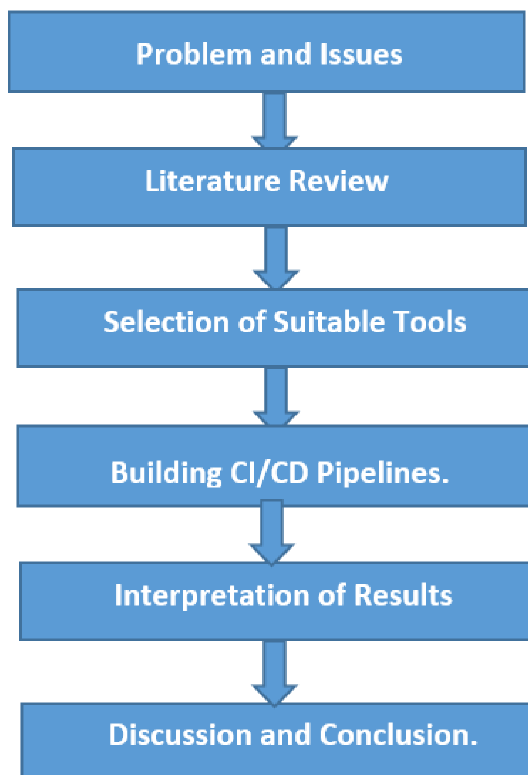

Figure 1: Flowchart illustrating the methodology

1.4 Scope of Study and Its Limitations

This study focuses on comparing three widely used and popular CI/CD tools which are GitHub Actions, Jenkins and GitLab CI. The thesis gives an emphasis on performance metrics such as build time, resource utilization and error rates. The evaluation focuses on building and deploying Docker container and deploying python Django applications on AWS EC2 instances, reflecting the modern software engineering workflows. The above analysis considers CPU usage, memory, disk I/O and error rate.

However, the study is limited to Django Python applications and results may differ in other programming languages and frameworks. It uses AWS to host using EC2 instances so performance may vary with other cloud providers as well as the instance types.

External factors like the latency of the network and resource contention could also to lead to the results being influenced. The study also does not dwell and evaluate all the features and functionalities of the CI/CD tools and only focuses on the basic performance metrics.

1.5 Organization of Thesis

This thesis work has been presented in following chapters:

Chapter 1 Gives an introduction about background, need of study, objectives of the research, methodology adopted in carrying out this research along with scope and limitations of the thesis.
Chapter 2 contains relevant literature reviews about CI/CD pipelines.
Chapter 3 Includes brief description about the pipelines setup.
Chapter 4 Deals with the obtained results and outputs along with the discussion and interpretation of the results.
Chapter 5 Incorporates the conclusion and recommendation for further study.
Reference for different relevant literature are listed at the end.

2 Literature Review

The landscape of the continuous integration and continuous delivery (CI/CD) has witnessed rapid and quick evolution with numerous tools emerging in the market trying to streamline the software development and deployment workflows. With tools like Jenkins and GitLab CI and GitHub Actions gaining significant attention with their unique features and performance along with their usability in diverse environments. This section of the literature review explores the existing studies conducted on this tools with focus on their strengths limitations and suitability in different scenarios.

2.1 Evolution of CI/CD Practices and Industry Adoption

The initiation of Continuous Integration (CI) can be traced towards the early 2000s which coincided with the rise of Agile methodologies. Studies the principles of CI bases that, emphasizing frequent code commits, rapid feedback loops and automated testing. Introduction of Continuous Deployment (CD) developed the principles much further with focus on automation of delivery pipelines with minimal human intervention (Flower, 2024).The shift on the process highlighted the emphasis on operation efficiency and responsiveness to the needs and demands of the user.

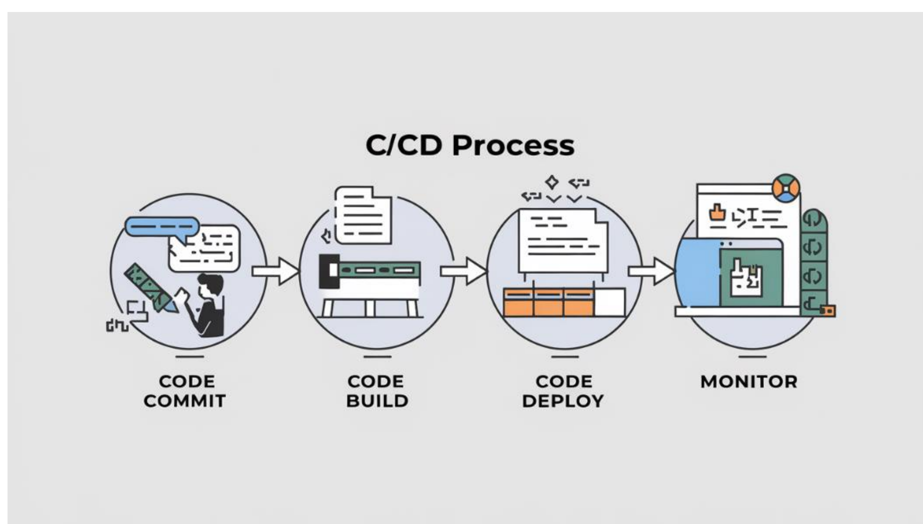

Figure 2: Illustration of the CI/CD process (Sławiński, 2023) .

CI/CD practices gained momentum with advent of the DevOps movement which advocated the unification of development teams. By bridging the gap between different functions of operation and development CI/CD pipelines emerged as one of the main components of the DevOps ecosystem.

CI/CD pipelines incorporation into the software engineering workflows is in alignment of the principles and adaptability of rapid delivery. Pipelines enable teams to integrate small incremental changes into the codebase which helps in frequent releases. The nature of Agile methodology which is iterative is complemented by the CI/CD automation capabilities which ensures streamlined testing and deployment process (Lwakatare et al., 2015).

CI/CD adoption is further enhanced by DevOps which promotes a culture of shared responsibility for application delivery. Tools and practices that are associated with CI/CD helps in reducing friction among conflicts and improved system stability. The increasing demand for automation in software development has steamrolled the creation of numerous CI/CD tools. GitLab CI, Jenkins and GitHub Actions have emerged out as the leading solutions providing diverse set of solutions.

Jenkins: One of the earliest CI/CD tools, Jenkins has a huge plugin ecosystem allowing customization. Despite its steep learning curve flexibility provided by the tool has made it one of the widely used tools in enterprise environments.

GitLab CI: Integrated into the GitLab Platform the tools simplify the workflows of CI/CD by incorporating version control, issue tracking and deployment capabilities.

GitHub Actions: It’s a newer entrant, GitHub Actions integration with GitHub is enticing to the developers as GitHub is the widely used source control. Its declarative workflow syntax and marketplace for actions, which are pre-built, help to facilitate rapid automation.

CI/CD pipelines despite their benefits pose challenges during implementation and smaller teams often struggle with the upfront investment required to configure and maintain the pipelines. In addition, the lack of metrics which are standardized for the evaluation of the CI/CD tools complicate the decision making process. Current trends focus on the need for simplified workflows, better scalability and more enhanced security features in CI/CD tools.

2.2 Introduction to Performance Metrics in CI/CD

CI/CD tools key point is their ability to deliver reliable fast and scalable pipelines along with minimizing the error rates and resource consumption. These are the foundation metrics used for evaluating a CI/CD tools. The section below delves deeper on the major performance metrics and their effect real world CI/CD pipelines.

Build Time

Build time directly impacts the developer's productivity so it's a critical metric. Feedback loop is shortened by faster build times which enables teams to identify and resolve issues promptly. Build times is the duration from the start of the build process till the creation of the deployable object. Build times metric is significant for agile teams as they practice frequent deployments so delays in build times can compound to substantial project shutdowns.

Build times are influenced by the codebase size so larger and complex codebases may typically result in longer build times. Pipeline configuration is the other factor which influences the build time inefficient steps in pipeline or poorly optimized build tools contribute to the delays. Tools that have caching mechanisms or execution capabilities that are parallel demonstrate performance which are superior like the pipeline caching feature present in GitLab CI reduces build durations.

Resource Utilization

Another key metric of evaluation is efficient resource utilization because efficient resource utilization ensures that CI/CD pipelines do not overburden infrastructure excessive operational costs.

Memory usage and CPU

Pipelines can be slowed down by high usage and other applications sharing the same environment can be affected. Efficient resource utilization ensures that CI/CD pipelines do not overburden the infrastructure or neither they incur excessive operational costs. If any tools require frequent read/write, they may experience bottlenecks in throughput environments' that are high.

The tools which are resource intensive can perform well in isolated environments' but they may struggle in cloud based environments' or shared environments'. Organizations often faced tradeoffs between the costs and performance when deploying the pipelines on platforms in cloud such as AWS or Azure.

In the context of tool specific observations, its been noted that GitLab CI demonstrated superior resource efficiency in the environments' that are containerized by leveraging Kubernetes and Docker on the other hand there is Jenkins which requires careful tuning if excessive resource consumption is to be avoided particularly when there are concurrent pipelines.

Error rates

They are used to measure reliability and efficiency while handling complex pipelines. There are different types of error in a CI/CD pipeline some of which are discussed ahead. Build Failures causes issues in dependency resolution and code compilation. Test failures errors occurs during automated execution of tests often due to incompatible frameworks or dependencies that are missing.

These are errors during packaging, staging or deployment and finally to production. Increased error rates also increase debugging time followed by delay releases it also diminish the confidence of developer in CI/CD system as error

handling capabilities are essential for maintaining seamless workflows. In context of comparison among tools Jenkins which allows integration of custom plugins which provides error detection and it demands steeper learning curve. The declarative workflow in the GitHub Actions simplifies error tracking but struggles in highly complex scenarios requiring fine tune controlling.

Scalability

Scalability of CI/CD tools becomes crucial as the size of projects grow in sizes and complexity. Some of the challenges of scaling CI/CD pipelines are that increased codebase size also lead to longer build times and test times in the scenario of high demand for concurrent builds in large teams requires tools with robust pipeline orchestration. Cloud based deployments also requires efficient scaling in order to keep the costs in control.

In performance under load it was noted that GitLab CI with its support for pipelines which are parallel and container orchestration seems too often outshine Jenkins and GitHub Actions and the pipeline requires substantial amount of manual configuration.

2.3 Existing CI/CD Tools

Jenkins: Legacy Leader in CI/CD

Jenkins launched in 2004 is an automation server based on open source and it has set its footprint in shaping the landscape of the CI/CD. Architecture of Jenkins is highly modular with support of over more than 1500 plugins that caters to use cases which are diverse in nature from simple pipelines builds and complex multi stage pipelines. Integration of various programming languages frameworks and deployment environments makes it a versatile choice for organizations.

Although Jenkins has its own strengths, Jenkins monolithic architecture presents scalability challenges particularly for application which are cloud native. Research has noted that Jenkins demands significant maintenance efforts to address plugin compatibility issues and frequent updates with high resource utilization

(Blueocean, 2025).However, because of its early entry into the landscape and its large community support and long standing reputation for reliability have secured it a place as preferred CI/CD tool for enterprises with dedicated DevOps teams.

Studies have highlighted Jenkins robustness in handling complex pipelines (Blueocean, 2025) . Jenkins excels in scenarios which require extensive customization such as building micro services architecture. However, this flexibility often comes at cost of setup complexity overhead in the operations leading to its deterrent for smaller teams.

GitLab CI: An integrated DevOps Ecosystem

GitLab CI is integrated seamlessly into the platform of GitLab and its offerings include unified environment for version control, CI/CD and monitoring. The integration in the platform eliminates the need for additional tooling, simplifying the process of CI/CD. It employs a YAML based system of configuration, which is required for defining policies allowing developers to specify different stages dependencies and workflows which are conditional.

GitLab CI's standout features include reliable support for containerization technologies such as Docker and Kubernetes. Capabilities of GitLab CI makes it a favors choice for cloud-native applications. The built in security and compliance tools such as vulnerability scanning and dependency checks addresses the growing requirement for secure software delivery.

Comparative analyses (Singh et al., 2019). reveal that GitLab CI outperforms Jenkins in terms of ease in setup and scalability. GitLab CI ability to handle concurrent pipelines with minimal changes in configuration has made it much more efficient for large scale projects. But the tool has steep learning curve with resource consumption which is high and is challenging for teams switching from similar simple CI/CD solutions.

GitHub Actions

GitHub Actions which was introduced in 2019 is a shift towards ease of adoption and rapid deployment and adoption of CI/CD tools. Its designed for integrating with GitHub repositories directly and using declarative syntax allowing developers to define workflows. Its marketplace has a vast library of pre-built actions which enables teams to seamlessly automate tasks such as linting testing and deployment without the requirement of extensive scripting.

Its strength lies on its seamless integration with the ecosystem of the GitHub and it’s easy to use interface. Its use has gained traction among small and medium sized teams who value simplicity and speed of adoption over extensive customization. Studies by (Mazrae, 2023) have shown that GitHub Actions decreases the time that’s required for setting up CI/CD pipelines in comparison to Jenkins and GitLab CI for projects that are being hosted on GitHub.

2.4 Comparison of Tools

Looking at Jenkins, GitLab CI and GitHub Actions across key performance matrices provides a clear picture of the strengths and limitations of the tools.

Build times

Although Jenkins has highly customizable design but it has slow out of the box configurations which lead to slower build times due to inefficient steps in pipeline or the tools which are poorly optimized resulting in delays but with proper configuration Jenkins seems to be on par with its competitors. GitLab CI stood out in the cases of large scale projects due to its faster build times along with its pipeline caching its stood out among other tool in the cases of large scale projects which requires frequent deployments. GitHub Actions performed well for project hosted on GitHub repositories its pre-built actions also simplified workflows which in turn reduced setup time but it struggles in with larger codebases then that of GitLab CI or Jenkins.

Error rates

The research showed that Jenkins excels in error detection because of its extensive plugin ecosystem as it allows for customizable error handling but its steep learning curve seems to pose teams unfamiliar with plugin configurations some issues. The robust error handling mechanisms in the GitLab ci is particular suited more for the contarized and kubernetes based environments with features like dependency checks and vulnerability scanning. Lastly GitHub Actions simplified the error tracking with its declarative workflows and its user friendly interface but it seemed to struggle in complex scenarios then GitLab ci or Jenkins where fine grained control was required.

Resource Utilization

Both Jenkins and GitLab CI are resource intensive in context of complex pipelines. Moderate resource efficiency is offered by GitHub Actions but it lacks detailed performance data for large workloads.

Ease of Use

In the context of ease of use, GitHub Actions leads because of its user-friendly setup and its integration with GitLab ecosystem. GitLab CI is the second best based on ease of use, while flexible Jenkins has the learning curve which is the steepest.

Scalability

In the context of handling concurrent pipelines with large scale deployments GitLab CI shines on the above context. Jenkins struggles with scalability in cloud environments', while GitHub Actions is suited for smaller projects (Singh et al., 2019).

Customization

When it comes towards the customization capabilities Jenkins remains unparalleled while flexibility with ease of use is balanced by GitLab CI. There are some constraints in advanced customization in GitHub Actions.

2.5 Challenges in Comparative Analysis

There aren't many standardized benchmarks which complicates the comparisons among the tools. Factors including application size, build complexity and different testing framework make generalizing it hard. However, some of the common CI/CD metrices that influences the pipeline which were listed above in the section 2.3 of the report were taken for comparative analysis of the three tools. Along with the rapid evolution of CI/CD tools, continuous evaluation is necessary to account for new features and updates.

Most studies have focused on Jenkins and GitLab CI which leaves GitHub Actions underreported in terms of quantitative analysis. Further Research aim should be having evaluation criteria which is standardized which allows for fair comparisons.

Scalability in Cloud Native workflows

The increased adoption of kubernetes in DevOps landscape further shows the need of CI/CD tools which are capable of managing the pipelines which are containerized also GitLab CI has shown promise in the domain but to validate the claims there needs to be further research with other tools including GitHub Actions and Jenkins.

Longitudinal Studies

As the CI/CD tools are rapidly evolving there is lack of studies to track the performance of these tools over a time. Studies would help organization for identifying the trends and predict the direction in which the industry is moving.

Future Research Directions and Practical implications

The need to address these research gaps is essential for advancing the good practices in CI/CD along with improvement of tool selection future research should focus on the below areas:

- **Scalability analysis**

  Development of universal benchmarks for resource utilization build times and error rates helps in enabling objective comparisons across tools it's also supportive cloud native architectures.

- **Diagnostics of error rate**

  The analysis of error rates in details including the cause as well as the resolutions will help organization in further optimization and downtime reduction.

Empirical studies on the tools provided insights which are valuable and delve into the strength and weakness of Jenkins, GitLab CI and GitHub Actions the leading tools in the market. In spite of this there remains a research gap in the areas such as standardized benchmarks optimization and scalability on cloud native environments addressing these gaps will not only advance the state of CI/CD research but also empower developers and organizations to make choices better in the software delivery process. The study also highlighted in how different scenarios each tool fitted.

3 Setting up Test Environment

For evaluation of the performance of the three tools mentioned above pipelines were created on all the three tools and executed using a Python Django application. The project was containerized using Docker and deployed on AWS EC2 instances. Timing metrics which included build times and execution duration for the task was collected through the Jenkins Timing Feature and through GitHub Actions logs and Pipeline details feature on the GitLab CI. The standardized approach provided a comparative analysis on the build efficiency resource utilization across three CI/CD pipelines.

3.1 Jenkins Setup

This section outlines approach employed for setting up and analyzing the continuous integration and deployment the goal of this experiment is to capture key time and performance metrics and compare Jenkins with other ci/cd tools such as GitHub Actions and GitLab CI the methodology is divided into different stage including environment setup, pipeline configuration and optimization strategies.

Setup of environment

The creation and execution of Jenkins pipeline by the use of following configuration and resources was done. The installation of Jenkins on Amazon EC2 instance with the use of Docker for streamline deployment was done for compatibility reasons.

Some of the essential plugins were add such as git and metrics plugin. Target project for the application was a python Django application which was hosted on GitHub. The repository requirements.txt was used for dependencies Docker configuration was used for containerization.

The pipeline of Jenkins was configured as to automate the test build and deployment process the pipeline was setup using the below steps. The first stage

was the pipeline creation stage which was used for navigating to the Jenkins dashboard and selecting new item and choosing the pipeline project type. Pipeline also took it script form the git the repository and its branch were specified which also had credentials for managing the private repositories. The Jenkins deployment file can be found on the Appendix section of the report.

The Jenkins file was created to define different stages in the pipeline it included stages checkout dependencies installation and test execution and building of Docker image lastly the deployment of Docker container on to a specified port. The summary execution report of the Jenkins pipeline can also be found on the Appendix section of this report.

3.2 GitLab CI Setup

The test stage in GitLab CI/CD pipeline covered approx. duration of 1 minute and 8 seconds. This stage likely involved running tests for verifying the functionality of Django application and ensuring its readiness for stages which are subsequent stages. Build stage which took 1 minute and 25 seconds included the Docker images creation as well as compiling the application code the tasks are resource intensive and critical for ensuring a reliable deployment artifact. The deploy stage of the pipeline was completed in 1 minute and 18 seconds the phase handled the deployment of dockerized application on an Aws EC2 instance it involved setting up necessary runtime environment and deployment of an application. The duration of pipeline execution for GitLab CI/CD was 3 minutes and 51 seconds which included 1 second of queuing time.

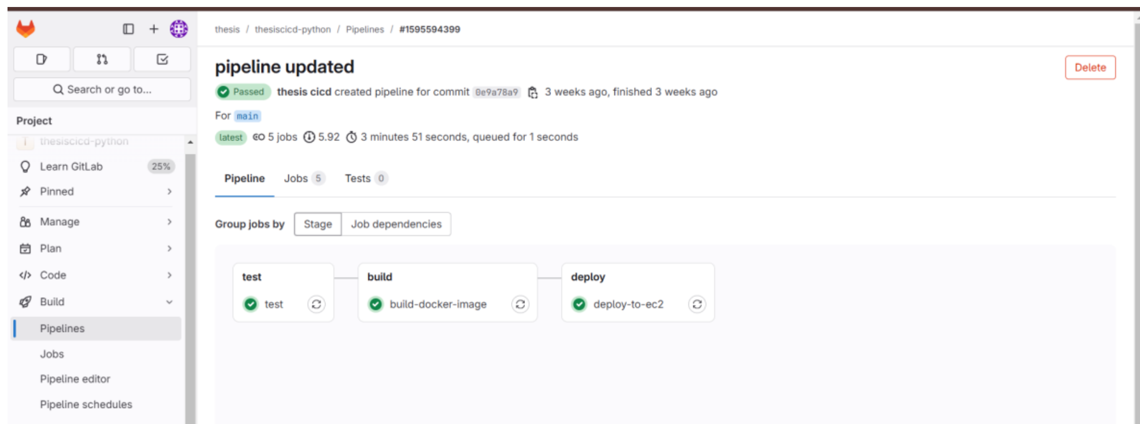

Figure 3: Three Stages of GitLab Ci pipeline.

Efficient execution of GitLab CI/CD pipeline completes all pipelines stages under four minutes the efficiency of pipelines is likely due to better caching mechanisms and environment preparation which was streamlined which minimizes the redundancies in the pipeline. The GitLab CI uses public runners which incurs delays due to dynamic provisioning and limited caching. The pipeline of GitLab are more flexible in caching Docker images and dependencies between stages the optimizations reduce build and deploy times significantly in comparison to GitHub Actions.

3.3 GitHub Action Setup

Setting up ci/cd pipeline is divided into three main stages testing stage then build and publish stage followed by the deployment stage. The testing stage validates the codebase integrity with the use of unit test the next stage build stage builds a Docker image and pushes it into the Docker hub lastly the deployment stage deploys the Docker container into an EC2 instance through ssh.

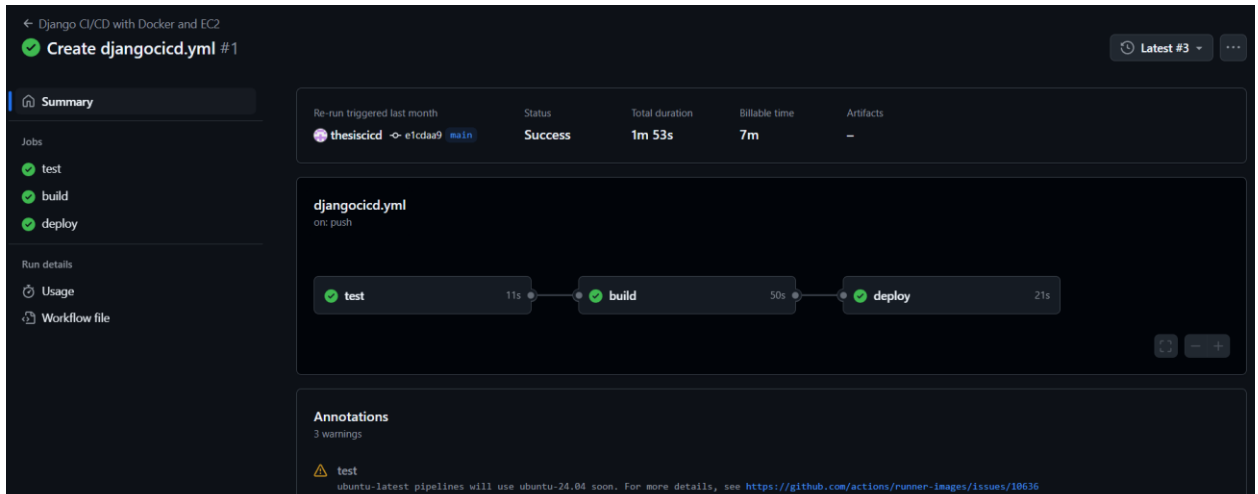

Figure 4: Figure showing the GitHub Actions pipeline interface with three stage

The pipeline is triggered on the following scenarios push trigger when a code is pushed into the main branch and also during the pull request trigger when a pull request targeting a main branch is created and updated the process ensures that every update to main branch or pull request undergoes the CI/CD process in complete.

Testing stage

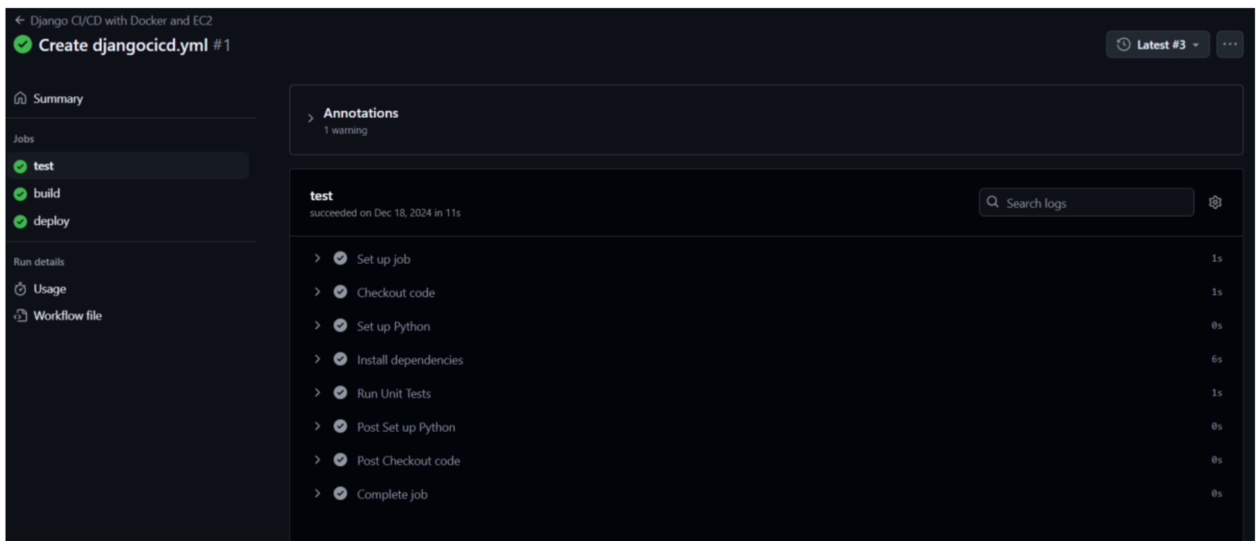

Figure 5: Test stage of the GitHub Actions pipeline

On to the first stage on CI/CD pipeline which is the testing stage which ensures that the application code passes the unit tests before proceeding to the build phase.

Build stage

The build stage of our code checkouts of the repository code sets up python version to 3.10 then install dependencies using pip install command the runs unit tests using python manage.py test.

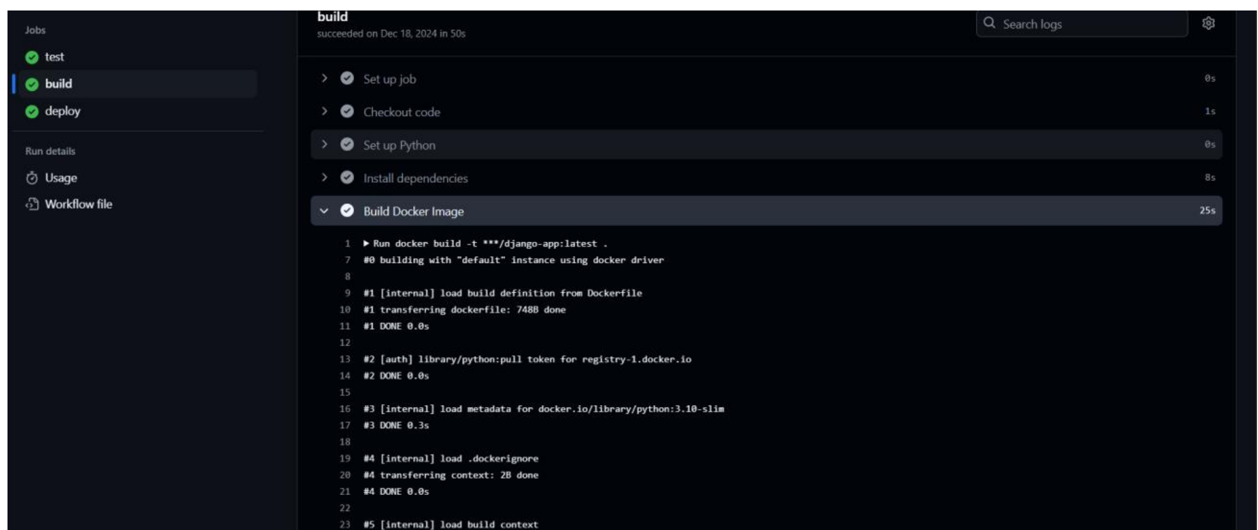

Figure 6: Build stage of the GitHub Actions pipeline

The purpose of the build stage of the pipeline in the GitHub Actions is to publish the image into the Docker hub. First checkout into the repository code then install dependencies and build the Docker image using Docker build uses GitHub Secrets then it will log onto the Docker hub then the Docker image is pushed to Docker hub.

Deployment stage

The purpose of this stage is to deploy the dockerized application to an EC2 instance. First we are logged to the EC2 instance via ssh using the private key which are stored securely in the secrets of GitHub Actions. The

deploy_django.sh script is transferred to the EC2 instance then the scripts executes the following into the EC2 instance which logs into the Docker hub and pulls the latest Docker image and stops any running container and then runs a new Docker container.

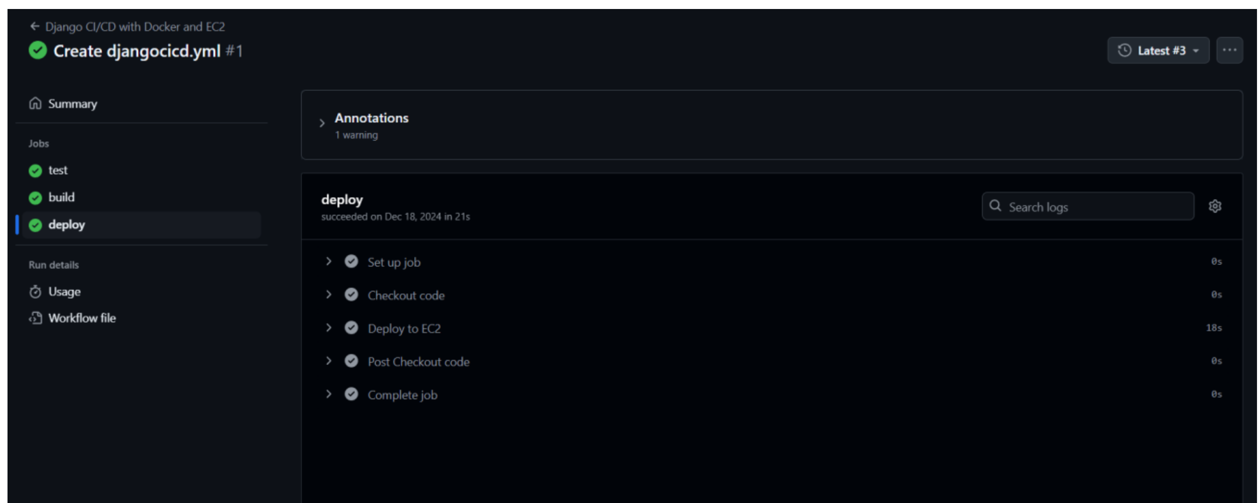

Figure 7: Deploy section of the GitHub Actions pipeline

Secrets and variables

The pipeline relied on GitHub secrets to securely store sensitive information:

- **DOCKER_USERNAME :** Docker Hub username and username and used in the Docker push step.
- **DOCKER_PASSWORD**: Docker hub password used for authentication.
- **EC2_HOST**: Public ip address or host name used in the EC2 instance.
- **EC2_USER**: The ssh username required for the EC2 instance.
- **EC2_KEY**: Used as private ssh key for connection into the EC2 instance

Configuring secrets in GitHub Actions

Start in the GitHub repository sections of settings onto subsection of secrets and variables, then proceed to the actions and finally new repository secrets was chosen and the above secrets was added.

Deployment Script

Script deploy_django.sh is copied to the EC2 instance during the deployment stage the script executes Docker hub login and pulls the latest Docker image of the Django application then we proceed to remove any existing applications in container then start to run the Docker container with the updated image as shown below.

```
#!/bin/bash

DOCKER_USERNAME=$1
DOCKER_PASSWORD=$2

# Authenticate with Docker Hub
echo $DOCKER_PASSWORD | docker login -u $DOCKER_USERNAME --
password-stdin

# Pull the latest Docker image
docker pull $DOCKER_USERNAME/django-finance:latest

# Stop and remove any existing container
docker rm -f django-finance || true

# Run the new container
docker run -d --name django-finance -p 8000:8000
$DOCKER_USERNAME/django-finance:latest
```

Best Practices of the Pipeline

The handling of secrets is done using GitHub secrets instead of hardcoded input as sensitive information such as ssh keys and Docker credentials need to be protected. Secrets should be rotated for maximum security. Non root user is used for running the application in the Docker container.

Stage Durations GitHub Actions

The CI/CD pipelines execution time which is divided into three stages test build and deploy the test stage of the pipeline took total of 11 seconds the stage included setup time dependency installation and running of unit tests the unit tests ran 10 tests in 3.884 seconds and the remaining 1 second are spent on checking out the code from repository and setting up python and installing its dependencies.

Step	Duration	Description
Set up job	0 seconds	Initial setup for the workflow environment.
Checkout code	0 seconds	Retrieves the latest code from the repository.
Set up Python	0 seconds	Configures the Python environment.
Install dependencies	4 seconds	Installs required Python packages from `requirements.txt`.
Build Docker Image	22 seconds	Creates a Docker image for the application based on the `Dockerfile`.
Log in to Docker Hub	0 seconds	Authenticates with Docker Hub using the stored credentials.
Push Docker Image	13 seconds	Pushes the built Docker image to the Docker Hub registry.
Post Set up Python	0 seconds	Cleans up after Python environment setup.
Post Checkout code	0 seconds	Cleans up after code checkout.
Complete job	0 seconds	Finalizes the build job.

Figure 8: Execution duration for the different steps of GitHub Actions pipeline

Build stage of the pipeline took 50 seconds with the Docker build stage taking bulk of the time as the base of the image and application dependencies are fetched and configured Docker catches the previously build repositories but after the build is complete the image is pushed to Docker hub. Total duration spent by our pipeline of GitHub Actions in deployment stage is 21seconds it takes a few moments for establishing secure connection with EC2 instance. Transfer and execution of deploy_django.sh script took most of the time on the EC2 instance which accounts for most of the time.

4 Results

This section discusses the observations for the CI/CD pipelines on Jenkins, GitHub Actions and GitLab CI. The execution time that were measured by deploying a Django application on to the EC2 instance provided us the results which showed the approx. time for Jenkins was 52 seconds followed by GitHub Actions at approx. of 1 minute 53 seconds and GitLab ci at 3 minutes 51 seconds the sections delves on to the detailed analysis focusing on execution time resource utilization and will also evaluate the reason behind those disparities

Table1. Comparative Analysis of the three CI/CD tools.

CI/CD Tool	Build Time	Resource Utilization.	Error Rate	Usability
Jenkins	52 seconds	Moderate	Low	Moderate
GitLab CI	3minute 51 seconds	High	Low	Moderate
GitHub Actions	1minute 53 seconds	Low	Moderate	High

4.1 Build Time

Jenkins

Fastest execution time was demonstrated by Jenkins among the three tools with its average build time being approximately 53 seconds. The efficiency of it can be attributed to its deployment on EC2 instances which are dedicated which in turn provided optimized job execution with caching strategies. The robust task isolation of Jenkins on the worker nodes minimized overhead and expedited builds. Its architecture allowed it to handle complex workflows without slowdowns which are significant in nature.

GitLab CI

GitLab CI had longest build time at 3 minute 51 seconds. The factors that caused it included less optimized configurations on certain workflows and higher overhead due to job orchestration and caching strategies which are less efficient. Using the parallel job execution and leveraging the advanced pipeline configuration could lead to reduction in build times significantly.

GitHub Actions

It has built time of approximately 1 minute 53 seconds and GitHub Action was slower than Jenkins. The self-hosted runners introduced provisioning overhead in spite of that its tight integration with repositories in GitHub streamlined other aspects of the build process. Despite slow build times of GitHub Actions flexibility and extensibility aspect of it make it practical choice for teams prioritizing ease of use.

4.2 Resource Utilization

Jenkins

Jenkins demonstrated efficient resource utilization which benefitted from its configurations on the dedicated EC2 instances it enabled resource management which resulted in minimal consumption and reduced build times along with the job queuing and caching mechanisms the ability to offload specific tasks to worker nodes contributed to its efficient resource usage.

GitLab CI

GitLab CI consumed more resources in comparison to other tools the high resource demand appeared due to its orchestration on containerized environments which although robust requires significant computational overhead. High levels of resource consumption by it is often justified in scenarios which requires high level of pipeline complexity.

GitHub Actions

Moderate resource usage was exhibited by GitHub Actions as its reliance on cloud hosted runners ensured balanced consumption. It also showed lack of efficiency in dedicated environment in comparison to Jenkins. The temporary nature of the runners of the GitHub Actions also showed strain on resources during high demand periods.

4.3 Error Rates

Jenkins

It showed low error rates which can be attributed to its ecosystem which is mature along with its customizable plugins which in turn support advanced error detection and handling. In turn this features make it highly reliable in terms of complex workflows in the scenario where the provided configurations are well maintained.

GitLab CI

Low error rates were recorded which reflects its robust error detection mechanisms especially in contarized and parallized workflows. Resolving errors can involve navigating its more intricate configurations.

GitHub Actions

It showed moderate error rates with its declarative workflows simplify error tracking they require fine tuning in scenarios which are complex to ensure reliability. In scenarios where there is misconfiguration in workflows or any compatibility issues with external dependencies can lead to increased error rates.

4.4 Scalability

Jenkins

Jenkins demonstrated strong scalability because of its highly customizable architecture and ability to handle concurrent builds efficiently in well tune environments. It also supported distributed builds which allowed teams to scale horizontally as demands for project increase.

GitLab CI

It showed to be highly scalable in container environments due to its extensive support for Kubernetes and Docker orchestration and also its robust pipeline configurations provided ability for supporting parallel workflows which in turn make it ideal for large complex projects but its higher resource consumption and build times which are slower may necessitate significant infrastructure optimizations.

GitHub Actions

The good scalability for teams leveraging GitHub repositories. Its cloud based runners adapt well for increasing workloads though dynamic provisioning which can become bottleneck under heavy load. Although for large scale projects additional infrastructure investments may be required to maintain the performance.

4.5 Job Execution and Parallelization

Jenkins

Jenkins excelled in this metric due to its advanced configurations for parallel job execution and job queuing which is efficient and its well-tuned caching mechanisms are design as modular which allowed multiple tasks to run concurrently with waiting times which are minimal. It also supports distributed builds which can be scaled horizontally for greater efficiency.

GitLab CI

GitLab CI provided support on parallel job execution but is less optimized than Jenkins but with high resource demands and caching mechanisms which results in longer waiting times. In spite of that its tight integration with container orchestration platforms like that of Kubernetes makes it highly adaptable for workflows requiring significant parallelization.

GitHub Actions

GitHub Actions supported parallelization but it faced challenges in maximizing efficiency due its reliance on dynamic provisioning. Tasks often compete for resources under heavy workloads. However, it’s easy to configure workflows so as they provide a strong foundation for teams which are smaller with parellization needs which are minimal.

5 Discussions and Conclusions

The study analyzed the efficiency of GitHub Actions, Jenkins, and GitLab CI in depth through the deployment of the Django-based website using Ci/CD pipelines on Docker on Aws EC2 instances. Key performance metrics were used to evaluate the three tools with a main emphasis on the build time and other metrics including resource utilization and factors such as usability scalability were delved into for proper evaluation of the different tools.

The analysis reveals distinct performance differences between GitHub Actions, GitLab CI in the context of deploying Django application on to an EC2 instance. Each tool demonstrated different level of efficiency in terms of build time, resource usage and error rates so following recommendation can be based on the findings. The execution time of Jenkins was the fastest with efficient resource usage which made it optimal choice for organizations that required high performance and on self-hosted CI/CD environments. Jenkins is more suitable for scenario where one requires customizability and flexibility. Also self-hosting of Jenkins is preferred as it allows full control of the infrastructure and it helps in optimizing the CI/CD pipeline performance. Jenkins is suitable for legacy systems needing robust mature CI/CD solutions which is typically the case of complex workflows.

GitHub Actions, while slower then Jenkins, provides proper solution as cloud native solution that is directly integrated with GitHub repositories. It is suitable to the teams which prioritize ease of use and it provides cloud base automation without the overhead required to manage the server or infrastructure. It is also ideal for open source projects.

GitLab CI exhibited the slowest execution time but it offered the most powerful features. It is recommended to the teams which used GitLab as the primary source of control platform. It is also deeply integrated into the GitLab ecosystem and offers a seamless experience in managing code. CI/CD process of GitLab CI

is best for organization that requires self-hosted CI/CD infrastructure with features such as auto DevOps and Kubernetes integration.

Ultimately the choice of CI/CD tools is guided by the needs of the specific project, the setup of the infrastructure and workflow of developer. Also the size of the codebase along with evaluation of key metrics such as build times, resource usage and integration features provided by the platform will help the development teams to make informed decision according to their needs.

The research and results serve as practical guide to make informed decisions on the three different CI/CD tools based on the project requirements and sizes. Future research can explore different functionalities provided by three tools in more depth with more exploration of additional factors including security and cost analysis which will help to make a proper decision.

References

1. Blue ocean (2025) *(PDF) enhancement of CICD pipelines with Jenkins Blueocean*. Available at: https://www.researchgate.net/publication/326555458_Enhancement_of_CICD_Pipelines_with_Jenkins_BlueOcean (Accessed: 17 January 2025).
2. Elazhary, O. *et al.* (2022) ‘Uncovering the benefits and challenges of continuous integration practices’, *IEEE Transactions on Software Engineering*, 48(7), pp. 2570–2583. doi:10.1109/tse.2021.3064953.
3. Flower, M. (2024) *Continuous integration*, *martinfowler.com*. Available at: https://martinfowler.com/articles/continuousIntegration.html (Accessed: 10 November 2024).
4. Hilton, M. *et al.* (2016) ‘Usage, costs, and benefits of continuous integration in open-source projects’, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 426–437. doi:10.1145/2970276.2970358.
5. Kinsman, T. (2021) *How do software developers use github actions to automate their workflows? | IEEE conference publication | IEEE xplore*. Available at: https://ieeexplore.ieee.org/document/9463074 (Accessed: 17 January 2025).
6. Lwakatare, L.E., Kuvaja, P. and Oivo, M. (2015) ‘Dimensions of DevOps’, *Lecture Notes in Business Information Processing*, pp. 212–217. doi:10.1007/978-3-319-18612-2_19.
7. Mazrae, P.R. (2023) *On the use of github actions in software development ...* Available at: https://pooya-rostami.github.io/papers/ICSME_2022_OnTheUseOfGitHubActionsInSoftwareDevelpmentRepositories.pdf (Accessed: 17 January 2025).
8. Sagar KhillarSagar Khillar is a prolific content/article/blog writer with a knack for crafting compelling content that captures the reader’s attention and drives engagement. He has that urge to research on versatile topics and develop high-quality content (2021) *Difference between github actions and Jenkins*, *Difference Between*. Available at: https://www.differencebetween.net/technology/difference-between-github-actions-and-jenkins/ (Accessed: 04 January 2025).
9. Shahin, M. (2015) ‘Architecting for DevOps and continuous deployment’, *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*, pp. 147–148. doi:10.1145/2811681.2824996.
10. Shahin, M., Ali Babar, M. and Zhu, L. (2017) ‘Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices’, *IEEE Access*, 5, pp. 3909–3943. doi:10.1109/access.2017.2685629.
11. Silva, R.B. and Bezerra, C.I. (2020) ‘Analyzing continuous integration bad practices in closed-source projects’, *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pp. 642–647. doi:10.1145/3422392.3422474.

12. Sławiński, T. (2023) *CI/CD pipelines: A comprehensive guide*, *Solidstudio*. Available at: https://solidstudio.io/blog/ci-cd-pipelines (Accessed: 17 January 2025).
13. Singh, C. *et al.* (2019) 'Comparison of different CI/CD tools integrated with cloud platform', *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)* [Preprint]. doi:10.1109/confluence.2019.8776985.
14. Vasiliev, D. (2024) *Continuous Integration Tools for DevOps - Jenkins vs. GitLab CI vs. Github Action*, *Attract Group*. Available at: https://attractgroup.com/blog/continuous-integration-tools-for-DevOps-jenkins-vs-GitLab-ci-vs-github-action/?utm_source=chatgpt.com (Accessed: 01 January 2025).
15. Zaid, Asad (2023) *Continuous delivery: Reliable software releases through build, test, and Deployment Automation* [Preprint]. doi:10.31219/osf.io/ksj5q

Appendix 1

Jenkins Deployment file

```
pipeline {
   agent any

   environment {
      DOCKER_IMAGE = "thesiscicd/thesiscicd-jenkins:latest" // Replace with
your Docker image name
   }

   stages {
      stage('Checkout Code') {
         steps {
            echo 'Checking out code...'
            checkout scm
         }
      }

      stage('Set Up Python') {
         steps {
            echo 'Setting up Python...'
            sh '''
            python3 -m venv venv
            source venv/bin/activate
            python3 -m pip install --upgrade pip
            pip install -r requirements.txt
            python manage.py check
            '''
         }
      }

      stage('Build Docker Image') {
```

```
        steps {
            withCredentials([usernamePassword(credentialsId:
'dockerHubCredentials', usernameVariable: 'DOCKER_USERNAME',
passwordVariable: 'DOCKER_PASSWORD')]) {
                script {
                    echo "Docker Username: ${DOCKER_USERNAME}"
                    echo "Building Docker Image: ${DOCKER_IMAGE}"

                    // Build the Docker image
                    sh """
                    docker build -t ${DOCKER_IMAGE} .
                    """
                }
            }
        }
    }

    stage('Push Docker Image') {
        steps {
            withCredentials([usernamePassword(credentialsId:
'dockerHubCredentials', usernameVariable: 'DOCKER_USERNAME',
passwordVariable: 'DOCKER_PASSWORD')]) {
                script {
                    echo "Pushing Docker Image: ${DOCKER_IMAGE}"

                    // Log in to Docker Hub
                    sh """
                    echo "${DOCKER_PASSWORD}" | docker login -u
"${DOCKER_USERNAME}" --password-stdin
                    docker push ${DOCKER_IMAGE}
                    """
                }
            }
```

```
            }
        }

        stage('Deploy to EC2') {
            steps {
                withCredentials([
                    file(credentialsId: 'PRIVATE_KEY_PATH', variable:
'PRIVATE_KEY_PATH'),
                    string(credentialsId: 'EC2_HOST', variable: 'EC2_HOST'),
                    string(credentialsId: 'EC2_USER', variable: 'EC2_USER'),
                    usernamePassword(credentialsId: 'dockerHubCredentials',
usernameVariable: 'DOCKER_USERNAME', passwordVariable:
'DOCKER_PASSWORD')
                ]) {
                    script {
                        // Ensure private key has proper permissions
                        sh "chmod 600 ${PRIVATE_KEY_PATH}"

                        // Copy deployment script to EC2
                        sh """
                        scp -o StrictHostKeyChecking=no -i ${PRIVATE_KEY_PATH}
deploy_django.sh ${EC2_USER}@${EC2_HOST}:/home/${EC2_USER}/
                        """


                        // Run deployment script on EC2 with Docker credentials and
image name
                        sh """
ssh -o StrictHostKeyChecking=no -i ${PRIVATE_KEY_PATH}
${EC2_USER}@${EC2_HOST} << EOF
    echo "Running Deployment Script"
    chmod +x /home/${EC2_USER}/deploy_django.sh
    /home/${EC2_USER}/deploy_django.sh ${DOCKER_USERNAME}
${DOCKER_PASSWORD} ${DOCKER_IMAGE}
```

```
EOF
"""
                    }
                }
            }
        }
    }

    post {
        always {
            echo 'Cleaning up...'
            cleanWs() // Cleans up the workspace, including sensitive files
        }
        success {
            echo 'Deployment successful!'
        }
        failure {
            echo 'Deployment failed!'
        }
    }
}
```

Appendix 2

Jenkins Execution Summary Report

Started by user thesiscicd
Lightweight checkout support not available, falling back to full checkout.
Checking out git https://github.com/thesiscicd/django-project.git into /var/lib/jenkins/workspace/jenkins-python@script/ce9e24e83f750e8f4ff7e2c0a1616760f4238d829f2a945d760e242ea62ea2d7 to read Jenkinsfile
The recommended git tool is: NONE
using credential 5
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/jenkins-python@script/ce9e24e83f750e8f4ff7e2c0a1616760f4238d829f2a945d760e242ea62ea2d7/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/thesiscicd/django-project.git # timeout=10
Fetching upstream changes from https://github.com/thesiscicd/django-project.git
 > git --version # timeout=10
 > git --version # 'git version 2.40.1'
using GIT_ASKPASS to set credentials
 > git fetch --tags --force --progress -- https://github.com/thesiscicd/django-project.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/feature/jenkins^{commit} # timeout=10
 > git rev-parse feature/jenkins^{commit} # timeout=10
Checking out Revision 6a76118e6db28334dee921ec5e7cc9d160cdfee2 (refs/remotes/origin/feature/jenkins)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 6a76118e6db28334dee921ec5e7cc9d160cdfee2 # timeout=10
Commit message: "Update Jenkinsfile"
 > git rev-list --no-walk e6a69cb63eeb533e15f292c50cf9e4aa288bac22 # timeout=10
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/jenkins-python
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Declarative: Checkout SCM)
[Pipeline] checkout
The recommended git tool is: NONE
using credential 5
Cloning the remote Git repository
Cloning repository https://github.com/thesiscicd/django-project.git
 > git init /var/lib/jenkins/workspace/jenkins-python # timeout=10
Fetching upstream changes from https://github.com/thesiscicd/django-project.git
 > git --version # timeout=10
 > git --version # 'git version 2.40.1'
using GIT_ASKPASS to set credentials

```
 > git fetch --tags --force --progress -- https://github.com/thesiscicd/django-project.git
+refs/heads/*:refs/remotes/origin/* # timeout=10
 > git config remote.origin.url https://github.com/thesiscicd/django-project.git # timeout=10
 > git config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
Avoid second fetch
 > git rev-parse refs/remotes/origin/feature/jenkins^{commit} # timeout=10
 > git rev-parse feature/jenkins^{commit} # timeout=10
Checking out Revision 6a76118e6db28334dee921ec5e7cc9d160cdfee2
(refs/remotes/origin/feature/jenkins)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 6a76118e6db28334dee921ec5e7cc9d160cdfee2 # timeout=10
Commit message: "Update Jenkinsfile"
[Pipeline] }
[Pipeline] // stage
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withEnv
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Checkout Code)
[Pipeline] echo
Checking out code...
[Pipeline] checkout
The recommended git tool is: NONE
using credential 5
 > git rev-parse --resolve-git-dir /var/lib/jenkins/workspace/jenkins-python/.git #
timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/thesiscicd/django-project.git #
timeout=10
Fetching upstream changes from https://github.com/thesiscicd/django-project.git
 > git --version # timeout=10
 > git --version # 'git version 2.40.1'
using GIT_ASKPASS to set credentials
 > git fetch --tags --force --progress -- https://github.com/thesiscicd/django-project.git
+refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/feature/jenkins^{commit} # timeout=10
 > git rev-parse feature/jenkins^{commit} # timeout=10
Checking out Revision 6a76118e6db28334dee921ec5e7cc9d160cdfee2
(refs/remotes/origin/feature/jenkins)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 6a76118e6db28334dee921ec5e7cc9d160cdfee2 #
timeout=10
Commit message: "Update Jenkinsfile"
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Set Up Python)
[Pipeline] echo
Setting up Python...
[Pipeline] sh
+ python3 -m venv venv
+ source venv/bin/activate
++ deactivate nondestructive
```

```
++ '[' -n '' ']'
++ '[' -n '' ']'
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
++ '[' -n '' ']'
++ unset VIRTUAL_ENV
++ '[' '!' nondestructive = nondestructive ']'
++ VIRTUAL_ENV=/var/lib/jenkins/workspace/jenkins-python/venv
++ export VIRTUAL_ENV
++ _OLD_VIRTUAL_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
++ PATH=/var/lib/jenkins/workspace/jenkins-
python/venv/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
++ export PATH
++ '[' -n '' ']'
++ '[' -z '' ']'
++ _OLD_VIRTUAL_PS1=
++ PS1='(venv) '
++ export PS1
++ '[' -n /usr/bin/sh -o -n '' ']'
++ hash -r
+ python3 -m pip install --upgrade pip
Requirement already satisfied: pip in ./venv/lib/python3.9/site-packages (21.3.1)
Collecting pip
  Using cached pip-24.3.1-py3-none-any.whl (1.8 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 21.3.1
    Uninstalling pip-21.3.1:
      Successfully uninstalled pip-21.3.1
Successfully installed pip-24.3.1
+ pip install -r requirements.txt
Collecting django (from -r requirements.txt (line 1))
  Using cached Django-4.2.17-py3-none-any.whl.metadata (4.1 kB)
Collecting Pillow (from -r requirements.txt (line 2))
  Using cached pillow-11.0.0-cp39-cp39-manylinux_2_28_x86_64.whl.metadata
(9.1 kB)
Collecting django-jazzmin (from -r requirements.txt (line 3))
  Using cached django_jazzmin-3.0.1-py3-none-any.whl.metadata (6.2 kB)
Collecting django-crispy-forms (from -r requirements.txt (line 4))
  Using cached django_crispy_forms-2.3-py3-none-any.whl.metadata (5.1 kB)
Collecting crispy-bootstrap4 (from -r requirements.txt (line 5))
  Using cached crispy_bootstrap4-2024.10-py3-none-any.whl.metadata (2.3 kB)
Collecting paypalrestsdk (from -r requirements.txt (line 6))
  Using cached paypalrestsdk-1.13.3-py3-none-any.whl.metadata (1.0 kB)
Collecting asgiref<4,>=3.6.0 (from django->-r requirements.txt (line 1))
  Using cached asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)
Collecting sqlparse>=0.3.1 (from django->-r requirements.txt (line 1))
  Using cached sqlparse-0.5.3-py3-none-any.whl.metadata (3.9 kB)
Collecting requests>=1.0.0 (from paypalrestsdk->-r requirements.txt (line 6))
```

```
  Using cached requests-2.32.3-py3-none-any.whl.metadata (4.6 kB)
Collecting six>=1.0.0 (from paypalrestsdk->-r requirements.txt (line 6))
  Using cached six-1.17.0-py2.py3-none-any.whl.metadata (1.7 kB)
Collecting pyopenssl>=0.15 (from paypalrestsdk->-r requirements.txt (line 6))
  Using cached pyOpenSSL-24.3.0-py3-none-any.whl.metadata (15 kB)
Collecting typing-extensions>=4 (from asgiref<4,>=3.6.0->django->-r
requirements.txt (line 1))
  Using cached typing_extensions-4.12.2-py3-none-any.whl.metadata (3.0 kB)
Collecting cryptography<45,>=41.0.5 (from pyopenssl>=0.15->paypalrestsdk->-
r requirements.txt (line 6))
  Using cached cryptography-44.0.0-cp39-abi3-
manylinux_2_28_x86_64.whl.metadata (5.7 kB)
Collecting charset-normalizer<4,>=2 (from requests>=1.0.0->paypalrestsdk->-r
requirements.txt (line 6))
  Using cached charset_normalizer-3.4.0-cp39-cp39-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (34 kB)
Collecting idna<4,>=2.5 (from requests>=1.0.0->paypalrestsdk->-r
requirements.txt (line 6))
  Using cached idna-3.10-py3-none-any.whl.metadata (10 kB)
Collecting urllib3<3,>=1.21.1 (from requests>=1.0.0->paypalrestsdk->-r
requirements.txt (line 6))
  Using cached urllib3-2.2.3-py3-none-any.whl.metadata (6.5 kB)
Collecting certifi>=2017.4.17 (from requests>=1.0.0->paypalrestsdk->-r
requirements.txt (line 6))
  Using cached certifi-2024.12.14-py3-none-any.whl.metadata (2.3 kB)
Collecting cffi>=1.12 (from cryptography<45,>=41.0.5->pyopenssl>=0.15-
>paypalrestsdk->-r requirements.txt (line 6))
  Using cached cffi-1.17.1-cp39-cp39-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting pycparser (from cffi>=1.12->cryptography<45,>=41.0.5-
>pyopenssl>=0.15->paypalrestsdk->-r requirements.txt (line 6))
  Using cached pycparser-2.22-py3-none-any.whl.metadata (943 bytes)
Using cached Django-4.2.17-py3-none-any.whl (8.0 MB)
Using cached pillow-11.0.0-cp39-cp39-manylinux_2_28_x86_64.whl (4.4 MB)
Using cached django_jazzmin-3.0.1-py3-none-any.whl (2.1 MB)
Using cached django_crispy_forms-2.3-py3-none-any.whl (31 kB)
Using cached crispy_bootstrap4-2024.10-py3-none-any.whl (23 kB)
Using cached paypalrestsdk-1.13.3-py3-none-any.whl (23 kB)
Using cached asgiref-3.8.1-py3-none-any.whl (23 kB)
Using cached pyOpenSSL-24.3.0-py3-none-any.whl (56 kB)
Using cached requests-2.32.3-py3-none-any.whl (64 kB)
Using cached six-1.17.0-py2.py3-none-any.whl (11 kB)
Using cached sqlparse-0.5.3-py3-none-any.whl (44 kB)
Using cached certifi-2024.12.14-py3-none-any.whl (164 kB)
Using cached charset_normalizer-3.4.0-cp39-cp39-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (144 kB)
Using cached cryptography-44.0.0-cp39-abi3-manylinux_2_28_x86_64.whl (4.2
MB)
Using cached idna-3.10-py3-none-any.whl (70 kB)
```

Using cached typing_extensions-4.12.2-py3-none-any.whl (37 kB)
Using cached urllib3-2.2.3-py3-none-any.whl (126 kB)
Using cached cffi-1.17.1-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (445 kB)
Using cached pycparser-2.22-py3-none-any.whl (117 kB)
Installing collected packages: urllib3, typing-extensions, sqlparse, six, pycparser, Pillow, idna, charset-normalizer, certifi, requests, cffi, asgiref, django, cryptography, pyopenssl, django-jazzmin, django-crispy-forms, paypalrestsdk, crispy-bootstrap4
Successfully installed Pillow-11.0.0 asgiref-3.8.1 certifi-2024.12.14 cffi-1.17.1 charset-normalizer-3.4.0 crispy-bootstrap4-2024.10 cryptography-44.0.0 django-4.2.17 django-crispy-forms-2.3 django-jazzmin-3.0.1 idna-3.10 paypalrestsdk-1.13.3 pycparser-2.22 pyopenssl-24.3.0 requests-2.32.3 six-1.17.0 sqlparse-0.5.3 typing-extensions-4.12.2 urllib3-2.2.3
+ python manage.py check
System check identified some issues:

WARNINGS:
?: (urls.W005) URL namespace 'basic' isn't unique. You may not be able to reverse all URLs in this namespace

System check identified 1 issue (0 silenced).
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build Docker Image)
[Pipeline] withCredentials
Masking supported pattern matches of $DOCKER_PASSWORD
[Pipeline] {
[Pipeline] script
[Pipeline] {
[Pipeline] echo
Docker Username: thesiscicd
[Pipeline] echo
Building Docker Image: thesiscicd/thesiscicd-jenkins:latest
[Pipeline] sh
+ docker build -t thesiscicd/thesiscicd-jenkins:latest .
#0 building with "default" instance using docker driver

#1 [internal] load build definition from Dockerfile
#1 transferring dockerfile: 801B done
#1 DONE 0.0s

#2 [internal] load metadata for docker.io/library/python:3.10-slim
#2 DONE 0.7s

#3 [internal] load .dockerignore
#3 transferring context: 2B done
#3 DONE 0.0s

```
#4 [1/6] FROM docker.io/library/python:3.10-
slim@sha256:61912260e578182d00b5e163eb4cfb13b35fb8782c98d1df9ed58
4cec8939097
#4 DONE 0.0s

#5 [internal] load build context
#5 transferring context: 90.49MB 2.6s done
#5 DONE 2.7s

#6 [2/6] WORKDIR /app
#6 CACHED

#7 [3/6] RUN apt-get update && apt-get install -y     gcc     libpq-dev     && rm -rf
/var/lib/apt/lists/*
#7 CACHED

#8 [4/6] COPY requirements.txt /app/
#8 CACHED

#9 [5/6] RUN pip install --no-cache-dir --upgrade pip &&     pip install --no-
cache-dir -r requirements.txt
#9 CACHED

#10 [6/6] COPY . /app/
#10 DONE 2.8s

#11 exporting to image
#11 exporting layers
#11 exporting layers 1.1s done
#11 writing image
sha256:d907993d0b495c24c60327673afb6411029ddc184fbaa76cae16fc1ed4d
c0aa5 done
#11 naming to docker.io/thesiscicd/thesiscicd-jenkins:latest done
#11 DONE 1.1s
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Push Docker Image)
[Pipeline] withCredentials
Masking supported pattern matches of $DOCKER_PASSWORD
[Pipeline] {
[Pipeline] script
[Pipeline] {
[Pipeline] echo
Pushing Docker Image: thesiscicd/thesiscicd-jenkins:latest
[Pipeline] sh
Warning: A secret was passed to "sh" using Groovy String interpolation, which is insecure.
```

Affected argument(s) used the following variable(s):
[DOCKER_PASSWORD]
See https://jenkins.io/redirect/groovy-string-interpolation for details.
+ echo ****
+ docker login -u thesiscicd --password-stdin
WARNING! Your password will be stored unencrypted in /var/lib/jenkins/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
+ docker push thesiscicd/thesiscicd-jenkins:latest
The push refers to repository [docker.io/thesiscicd/thesiscicd-jenkins]
120c4cc6ea52: Preparing
6b4f89f0f361: Preparing
83fbf7289853: Preparing
df25a65d4880: Preparing
52510431889d: Preparing
597aba2d3779: Preparing
4694bafa367d: Preparing
15845fc8070f: Preparing
c0f1022b22a9: Preparing
597aba2d3779: Waiting
4694bafa367d: Waiting
15845fc8070f: Waiting
c0f1022b22a9: Waiting
52510431889d: Layer already exists
df25a65d4880: Layer already exists
83fbf7289853: Layer already exists
6b4f89f0f361: Layer already exists
597aba2d3779: Layer already exists
4694bafa367d: Layer already exists
15845fc8070f: Layer already exists
c0f1022b22a9: Layer already exists
120c4cc6ea52: Pushed
latest: digest:
sha256:edd3c14c06ab5f8d01dde0964e68e0da2e5eb94f12e8225eb68319c0c1c25a5d size:
2208
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy to EC2)
[Pipeline] withCredentials
Masking supported pattern matches of $PRIVATE_KEY_PATH or $EC2_HOST or $EC2_USER or $DOCKER_PASSWORD
[Pipeline] {
[Pipeline] script
[Pipeline] {
[Pipeline] sh
Warning: A secret was passed to "sh" using Groovy String interpolation, which is insecure.
Affected argument(s) used the following variable(s):
[PRIVATE_KEY_PATH]
See https://jenkins.io/redirect/groovy-string-interpolation for details.
+ chmod 600 ****

```
[Pipeline] sh
Warning: A secret was passed to "sh" using Groovy String interpolation, which is insecure.
                 Affected argument(s) used the following variable(s): [EC2_USER, EC2_HOST, PRIVATE_KEY_PATH]
                 See https://jenkins.io/redirect/groovy-string-interpolation for details.
+ scp -o StrictHostKeyChecking=no -i **** deploy_django.sh ****@****:/home/****/
[Pipeline] sh
Warning: A secret was passed to "sh" using Groovy String interpolation, which is insecure.
                 Affected argument(s) used the following variable(s): [EC2_USER, EC2_HOST, DOCKER_PASSWORD, PRIVATE_KEY_PATH]
                 See https://jenkins.io/redirect/groovy-string-interpolation for details.
+ ssh -o StrictHostKeyChecking=no -i **** ****@****
Pseudo-terminal will not be allocated because stdin is not a terminal.

A newer release of "Amazon Linux" is available.
  Version 2023.6.20241212:
Run "/usr/bin/dnf check-release-update" for full release and version update info
   ,     #_
   ~\_  ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
Running Deployment Script
WARNING! Your password will be stored unencrypted in /home/****/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
latest: Pulling from thesiscicd/thesiscicd-jenkins
Digest: sha256:edd3c14c06ab5f8d01dde0964e68e0da2e5eb94f12e8225eb68319c0c1c25a5d
Status: Image is up to date for thesiscicd/thesiscicd-jenkins:latest
docker.io/thesiscicd/thesiscicd-jenkins:latest
c0fb11b42351507bff55ae0e4099d547fc16057829e787bfb2b4b515f7f67989
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
```

```
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Cleaning up...
[Pipeline] cleanWs
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Deferred wipeout is used...
[WS-CLEANUP] done
[Pipeline] echo
Deployment successful!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```