Continuous Integration and Continuous Delivery (CI/CD) Pipeline

Setting Up a CI/CD Pipeline with GitHub, EC2, and Jenkins

Introduction:

In this report, I provided a thorough rundown of the Continuous Integration and Continuous Delivery (CI/CD) pipeline I created using GitHub, Amazon EC2 (Elastic Compute Cloud), and Jenkins. This pipeline is a crucial component of my Portfolio project, aiming to streamline development, testing, and deployment processes.

The pipeline serves as a pivotal component in the development life cycle of the Portfolio project, designed to enhance efficiency, reduce errors, and expedite the deployment of new features and bug fixes. Throughout this report, I will explain the rationale behind each tool selection and provide a detailed account of the configuration and integration process.

Source Code Repository (GitHub):

To kick off the CI/CD pipeline, I initiated this CI/CD pipeline by creating a source code repository on GitHub for my Portfolio project. GitHub was chosen as the platform for source code management due to its popularity, collaboration features, and its seamless integration with various CI/CD tools.

To set up the repository, I signed up for a GitHub account and created a new repository with the name 'Portfolio.' GitHub serves as the cornerstone for source code management. It provides a collaborative environment for storing, tracking, and sharing code changes

1. GitHub Account Setup:

Initiating the CI/CD pipeline began with the creation of a GitHub account. GitHub offers a user-friendly and widely-used platform for hosting and managing source code, making it a natural choice for your project.

A GitHub account allows you to create and manage repositories, collaborate with others, and leverage a range of version control and project management features.

2. Repository Creation:

After signing up, I initiated a new repository with a meaningful name and description that reflected my Portfolio project. Choosing a descriptive name helps team members and collaborators understand the repository's purpose easily.

The repository's description should provide additional context, summarizing the project's goals, features, and any special considerations.

3. Repository Settings:

I configured the repository's settings according to the project requirements, ensuring that access permissions and repository details were appropriately set. I also made use of

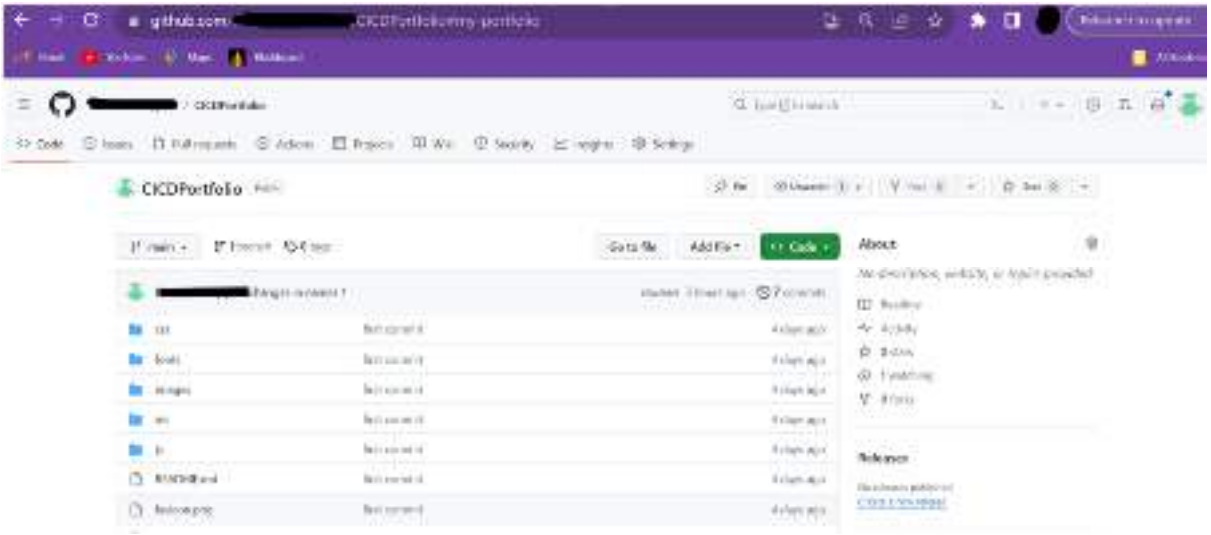
Gitbash, a command-line tool, to upload my Portfolio project files to the repository. Gitbash's command features simplified the process of pushing changes to the repository.

Configuring the repository's settings is essential to align with the project's requirements. These settings ensure that access permissions and other repository details are appropriately configured. Some key configuration steps include:

Access Permissions: You may set access permissions to control who can view, collaborate, or modify the repository. This is important for maintaining the security and integrity of the codebase.

Branch Protection: Branch protection rules can be established to prevent accidental changes to critical branches. This ensures that only authorized individuals can merge changes into specific branches.

Collaboration Features: Pull requests, bugs, and conversations are just a few of the collaboration tools that GitHub provides to improve code review and teamwork.



4. Branching and Version Control:

As the project evolved, I leveraged GitHub's branching and version control capabilities to manage changes effectively. Until they were ready for integration, branches allowed for the creation of new features or bug fixes without having an impact on the main codebase.

GitHub offers a powerful version control system with Git, allowing me to track changes, collaborate with team members, and manage my project's history effectively

5. Collaboration:

GitHub fosters collaboration by allowing multiple contributors to work on the project simultaneously. In addition to encouraging peer review, this guarantees that code changes are carefully reviewed before being merged into the main branch.

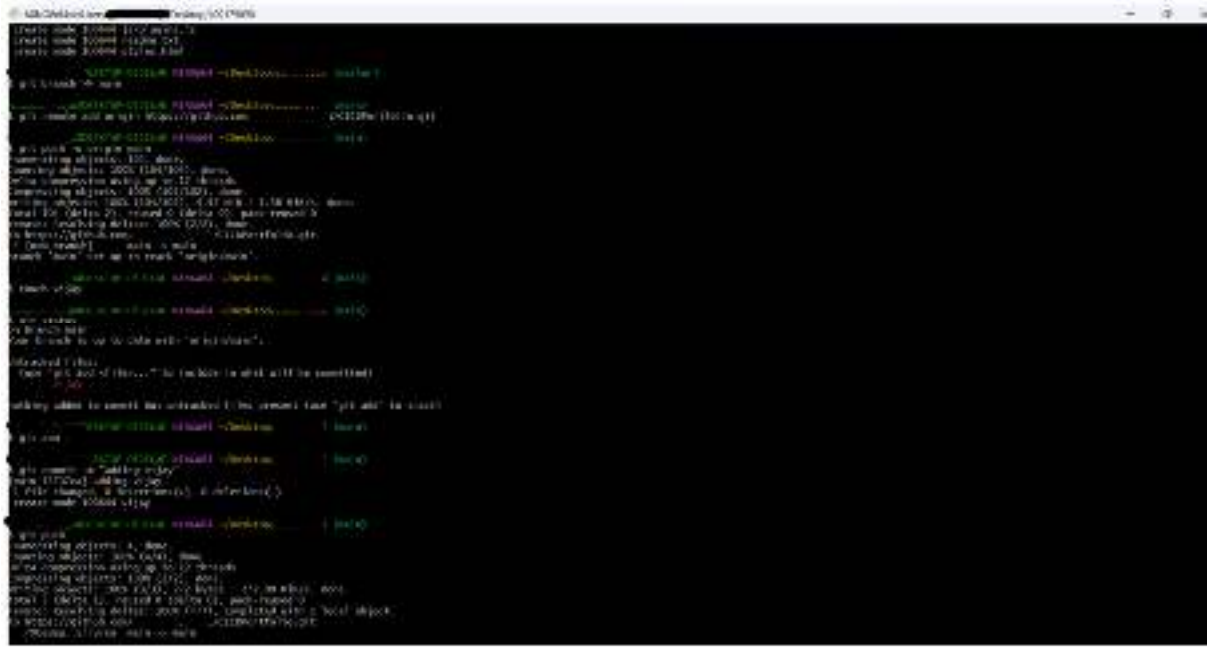
6. Security:

GitHub provides robust security features, including branch protection, code scanning, and access control. These features are vital for safeguarding the project's source code against unauthorized changes and security vulnerabilities.

7. Community and Integration:

GitHub boasts a vibrant community of developers, which can be a valuable resource for seeking help, sharing knowledge, and discovering open-source projects and tools.

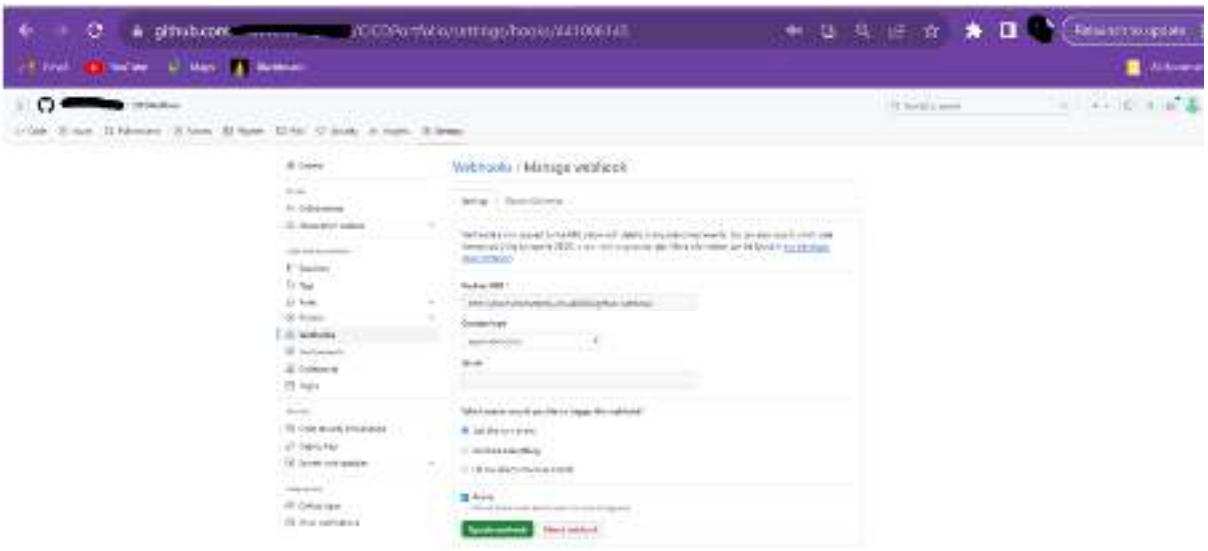
GitHub integrates seamlessly with a wide range of CI/CD tools, including Jenkins, which was a key consideration in your tool selection. This integration allows for automated builds and deployments triggered by code changes.



8. Webhooks:

To further automate the CI/CD pipeline, I added webhooks in the repository's settings. Webhooks enable automatic triggering of the CI/CD pipeline when code changes occur in the repository.

By configuring webhooks, I ensure that the development environment remains continuously updated, reducing manual intervention and improving overall efficiency in the pipeline.



- In summary, setting up a source code repository on GitHub is a foundational step in my CI/CD pipeline. It provides a secure, collaborative, and version-controlled environment for my project's code. Leveraging GitHub's features, i can streamline code development, facilitate collaboration, and ensure that code changes are thoroughly reviewed and tested before deployment. Furthermore, the development workflow is automated and the development environment is always up to date with the most recent code changes thanks to the interaction with other CI/CD tools via webhooks.

Automated Build and Testing Using Jenkins:

To automate the build and testing processes, I employed Jenkins, a popular CI/CD automation tool. Jenkins was installed on an Amazon EC2 instance, specifically a t2 medium instance as it offers a balance of compute resources for our CI/CD pipeline.

In my CI/CD pipeline, Jenkins functions as the main center for process automation related to build, testing, and deployment. It guarantees that code updates are adequately tested before being released and optimizes the development workflow. The setup involved various steps:

1. EC2 Instance Configuration: (Article, 2021)

To host Jenkins, I selected an Amazon Elastic Compute Cloud (EC2) instance, specifically choosing a t2 medium instance that offers a balanced level of computational power. This instance type provides enough resources to run Jenkins efficiently.

The selection of the EC2 instance's specifications is essential, as it directly impacts the performance and reliability of Jenkins. Factors to consider include CPU, memory, and storage capacity, all tailored to the project's requirements.

2. Jenkins Installation:

I installed Jenkins on the EC2 instance following the installation instructions provided by the official Jenkins documentation. Jenkins is a popular choice for CI/CD due to its extensibility and a wide range of available plugins.

Jenkins can be installed as a standalone service or as a Docker container, depending on your preferences and requirements. The installation process typically involves downloading and running a Java-based Jenkins package on the EC2 instance.

3. Configuration:

After installation, I configured Jenkins to build and test my project automatically. This included setting up build jobs, specifying build triggers (such as GitHub webhooks), and integrating with version control repositories.

Build Jobs: Jenkins allows you to create and configure build jobs, defining the steps necessary to build your project. You specify build triggers (such as GitHub webhooks) and set the project's source code repository.

Build Triggers: To ensure automation, build triggers are set up. This typically includes configuring GitHub webhooks to notify Jenkins when code changes are pushed to the GitHub repository. Jenkins can also use polling mechanisms to check for changes at regular intervals.

Version Control Integration: Integration with the version control system, in this case, Git, is fundamental. Jenkins can be set up to automatically pull the latest code from the GitHub repository when a build is triggered.

Plugin Installation: Jenkins provides a wide range of plugins to extend its capabilities. Depending on your project's requirements, we can install and configure plugins for additional functionality, such as code analysis, automated testing, and deployment.

4. Security Groups:

Security groups were configured to control incoming and outgoing traffic to the EC2 instance. This enhanced the overall security of the Jenkins server.

By defining security group rules, we can restrict access to the EC2 instance, allowing only trusted IP addresses and specific ports to connect. This helps prevent unauthorized access to Jenkins and secures your CI/CD pipeline.

5. Domain Name Assignment:

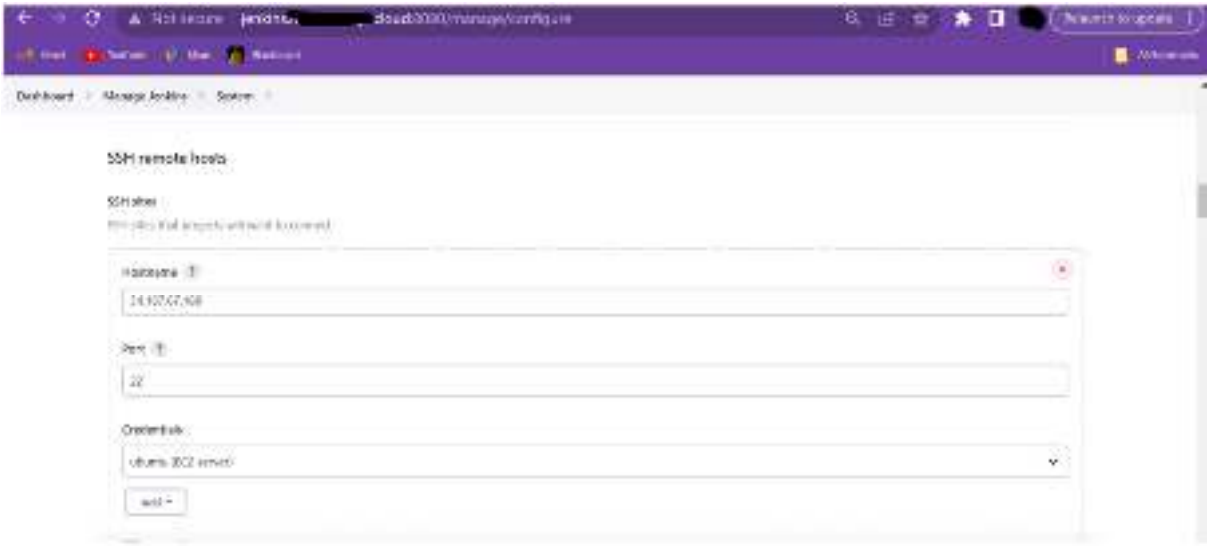
To make Jenkins accessible from the web, I assigned a domain name (jenkins.honneshraju.cloud) to the EC2 instance's public IP address. This step simplifies access to the Jenkins server and makes it accessible via a user-friendly URL.

6. Jenkins Configuration:

In the Jenkins dashboard, I accessed "Manage Jenkins" and configured build triggers by adding the "GitHub hook trigger for GITScm." This setting ensured that Jenkins automatically triggered builds upon code changes in the GitHub repository.

The Jenkins configuration also involves specifying which build agents or nodes to use for various tasks. This allows us to distribute build and test workloads efficiently across multiple nodes if needed.

7. SSH Remote Hosts: To enable Jenkins to connect to the EC2 instance securely, I added the EC2 instance as an SSH remote host in the Jenkins dashboard. This allowed Jenkins to execute commands remotely.



- Jenkins' automated build and testing process guarantees that code changes are completely verified and validated while streamlining the development workflow and minimizing manual involvement. Jenkins not only automates the compilation and build process but can also integrate various testing frameworks to check the code's quality and functionality before deployment. This process helps identify and resolve issues early in the development cycle, contributing to the overall reliability and quality of the Portfolio project.

Deployment to a Staging Environment Using AWS EC2:

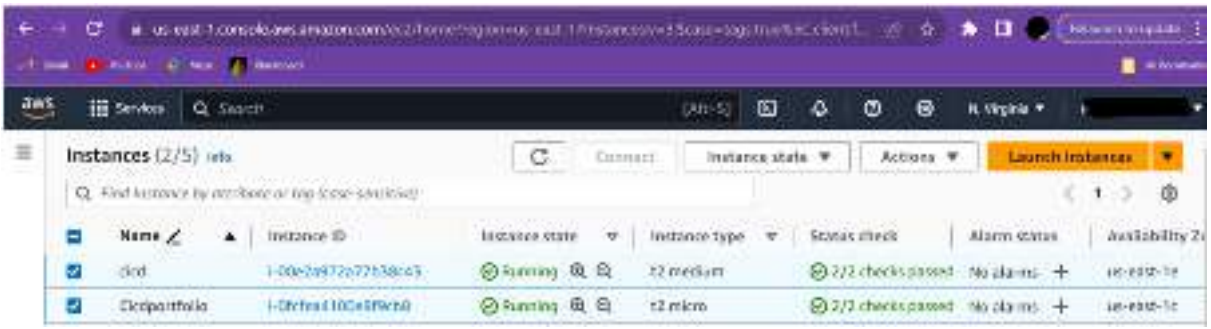
The next phase in my CI/CD pipeline involved deploying my Portfolio project to a staging environment hosted on an AWS EC2 instance. I chose another EC2 t2 Micro instance for hosting the staging environment to keep the production and staging environments separate. The following steps were taken:

1. EC2 Instance Setup:

The first step involved launching a separate Amazon Elastic Compute Cloud (EC2) instance dedicated to serving as the staging environment. Choosing a different instance from the Jenkins server ensures a clear separation between the development and deployment environments. This instance is separate from the Jenkins server, keeping development and deployment isolated.

The selection of a t2 Micro instance for staging can be a cost-effective choice as these instances provide a baseline level of CPU performance while being eligible for the AWS Free Tier, which can be beneficial for smaller-scale projects and testing purposes.

During the setup, it's crucial to select an appropriate Amazon Machine Image (AMI) that aligns with your project's requirements, here I selected Ubuntu operating system and pre-installed software packages.



2. Integration with Git:

To make sure the staging EC2 instance gets code updates from the GitHub repository automatically, I integrated it with the repository. The staging environment is always running the most recent code thanks to this integration.

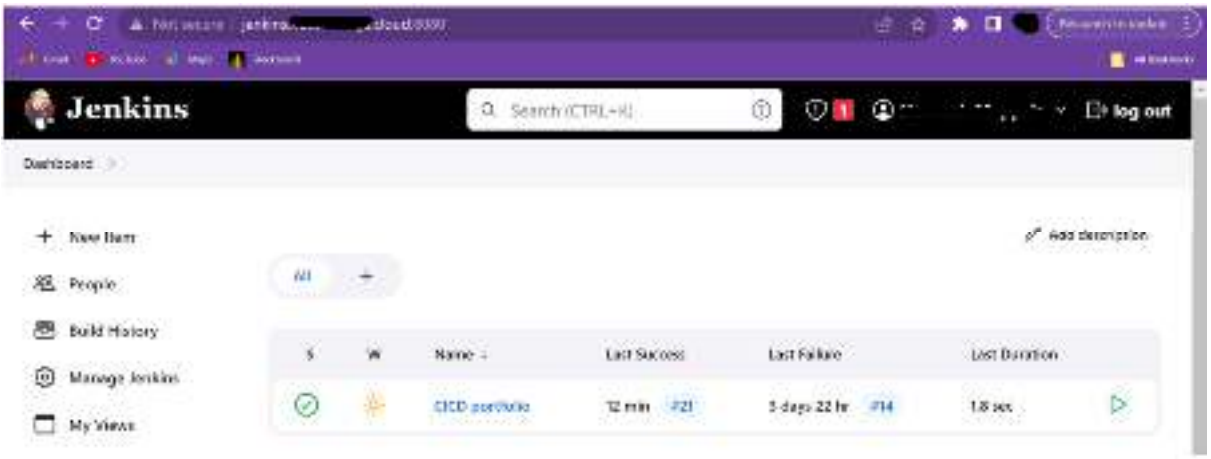
Git hooks or other automation mechanisms can be used to trigger updates on the staging EC2 instance when new commits are pushed to the GitHub repository. This automation minimizes the manual intervention required to keep the staging environment current.

3. Jenkins Integration:

I configured Jenkins to deploy the project to the staging environment, ensuring that code changes are automatically deployed when the pipeline is triggered. Jenkins can trigger deployment jobs based on certain conditions, such as successful builds.

Jenkins can be configured to execute deployment jobs based on specific conditions. For example, a deployment job may be triggered only when the build is successful, ensuring that only reliable code changes are deployed to the staging environment.

Properly configuring the Jenkins server to deploy to the staging EC2 instance involves specifying the target environment, authentication credentials, and scripts or commands necessary for the deployment. This may include copying files to the EC2 instance, setting up databases, configuring web servers, and more, depending on the specific requirements of the project.



4. Installation of Git:

To facilitate version control, I installed Git on the staging EC2 instance. This allowed for code repository cloning from GitHub and efficient code management within the staging environment. For instance, I ran the following commands on an EC2 instance running Ubuntu:

{*(Sudo apt-get update) *(Sudo apt-get install git)}

5. Installation of Apache2:

I installed Apache2 on the EC2 instance to set up a web server for the staging environment. This web server provided a platform for testing the application and allowing stakeholders to access the staging environment for validation. For instance, I ran the following commands on an EC2 instance running Ubuntu:

{*(Sudo apt-get update) *(Sudo apt-get install apache2)}

6. Security Group Configuration:

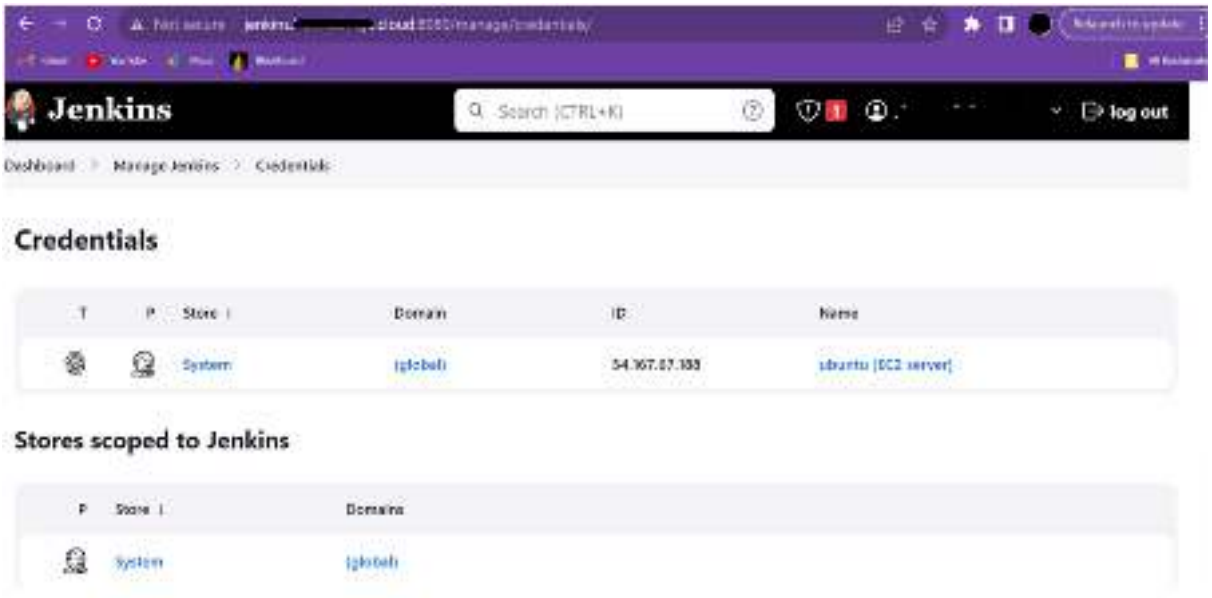
Security groups were added to the EC2 instance to manage network access, ensuring that the staging environment was secure. Security groups are an integral part of the AWS ecosystem. They act as virtual firewalls, controlling inbound and outbound traffic to EC2 instances.

In the context of staging environment deployment, security groups should be configured to restrict access to the EC2 instance to only the necessary IP addresses and ports. This helps in enhancing the security of the staging environment by minimizing potential attack vectors.

7. Jenkins Credentials:

In the Jenkins dashboard, under the "Manage Jenkins" section, credentials should be added to allow Jenkins to access the staging environment's EC2 instance securely.

These credentials may include SSH keys, access tokens, or other authentication methods required for Jenkins to establish a secure connection and perform deployment tasks on the staging EC2 instance



8. Development Workflow:

In Visual Studio Code or a similar development environment, I continued to make changes to the code. To push these changes to the staging environment, I executed Git commands like "git add," "git commit," and "git push."

9. Jenkins Verification:

After pushing code changes to the GitHub repository, monitoring Jenkins is vital to verify the success of the deployment process. Jenkins provides detailed logs and feedback on the deployment job's progress, allowing you to identify and rectify any issues that may arise during deployment.

Continuous monitoring ensures that changes are seamlessly integrated into the staging environment, and any deployment failures are quickly addressed.

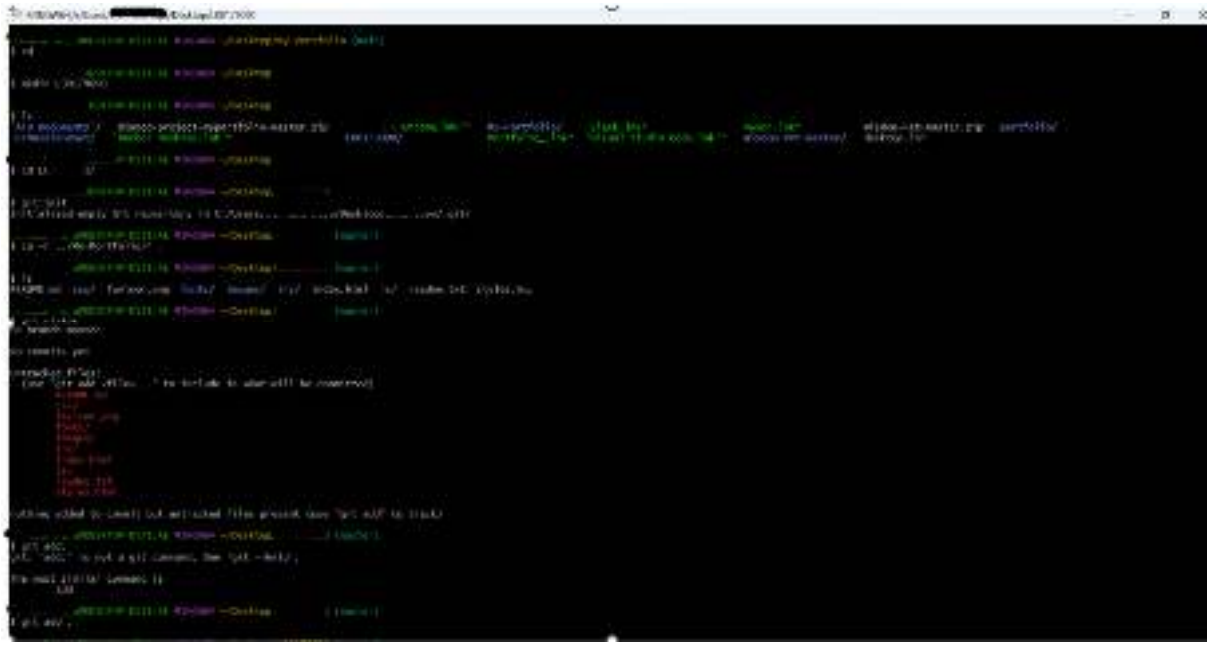
10. Domain Assignment:

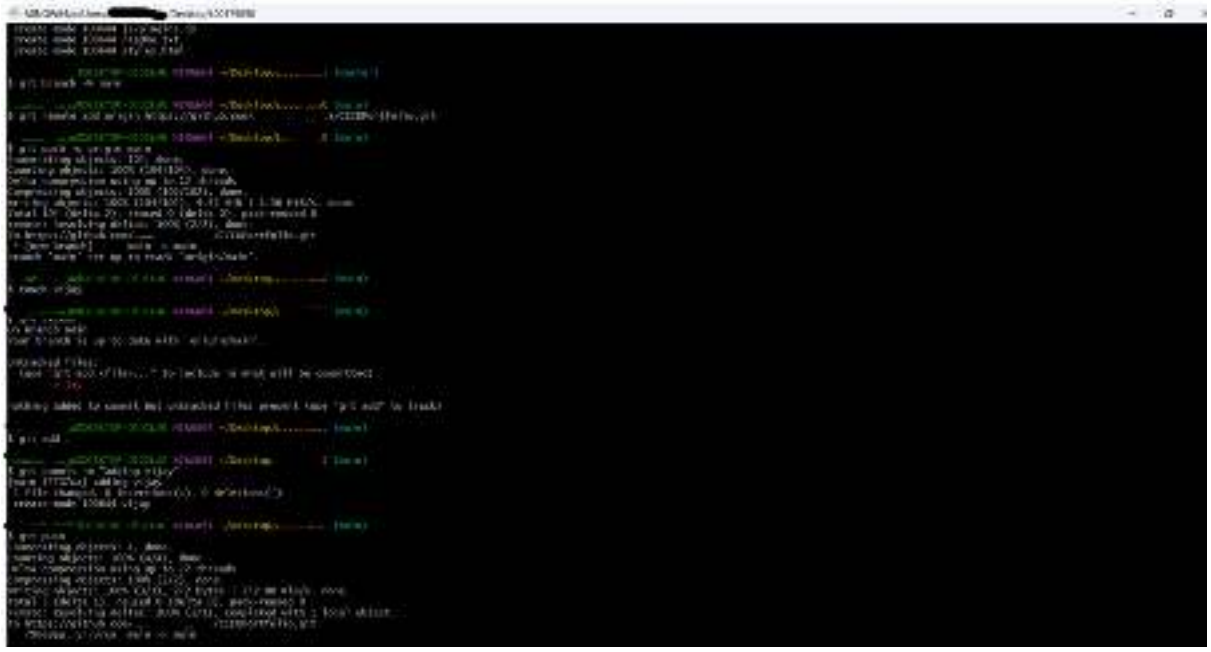
To provide user-friendly access to the staging environment for testing and validation, a domain name (atu.honneshraju.cloud) is assigned to the EC2 instance's public IP address. This makes it easy for team members, stakeholders, and quality assurance teams to access the staging environment via a memorable and user-friendly URL.

- The deployment to a staging environment using AWS EC2 is an integral part of your CI/CD pipeline. It guarantees that code modifications are verified and extensively tested prior to being pushed into the live environment. By installing Git and Apache2 on the EC2 instance, you create an environment where version control and web server capabilities are readily available for efficient testing and validation of your application. This approach enhances the reliability and stability of your Portfolio project while reducing the risk of introducing bugs or issues to the live system.

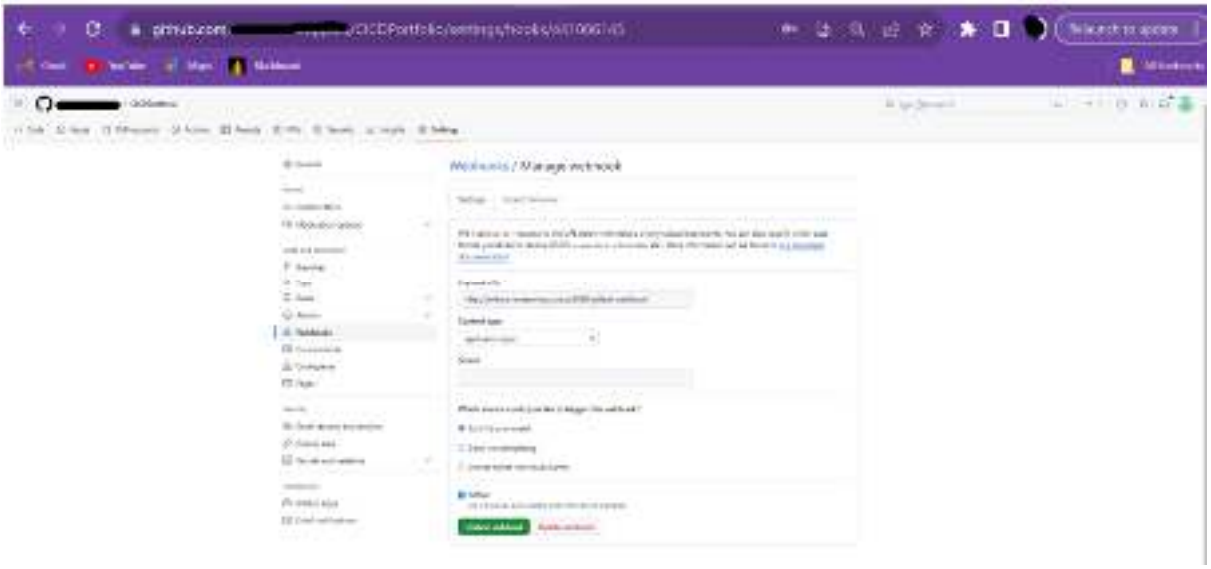
Results:

1. **Repo creation and uploading files to GIT**

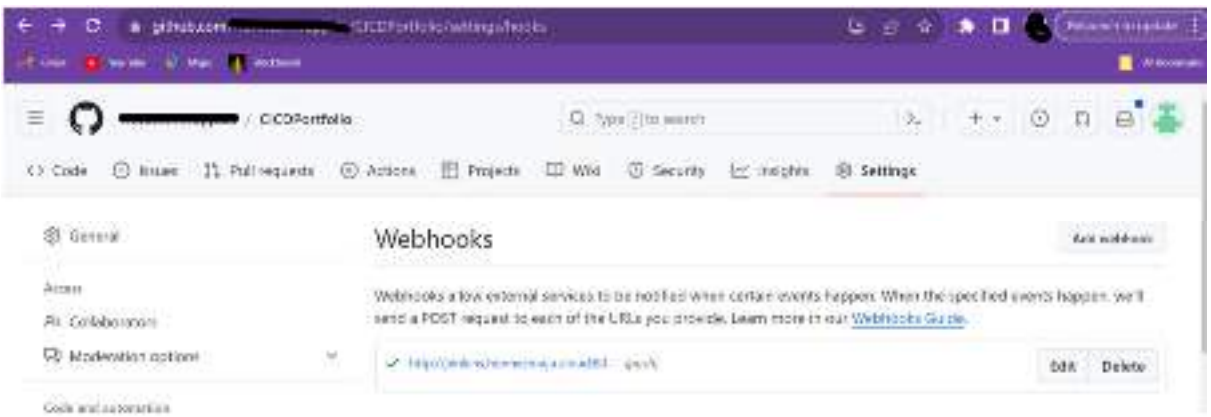


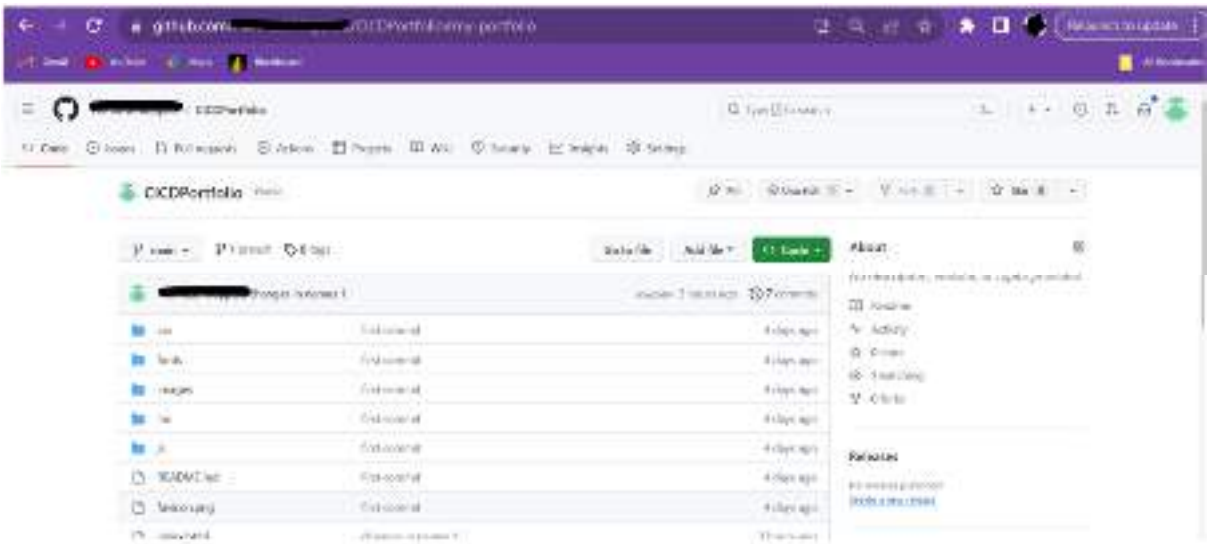


2. GitHub settings and Webhook

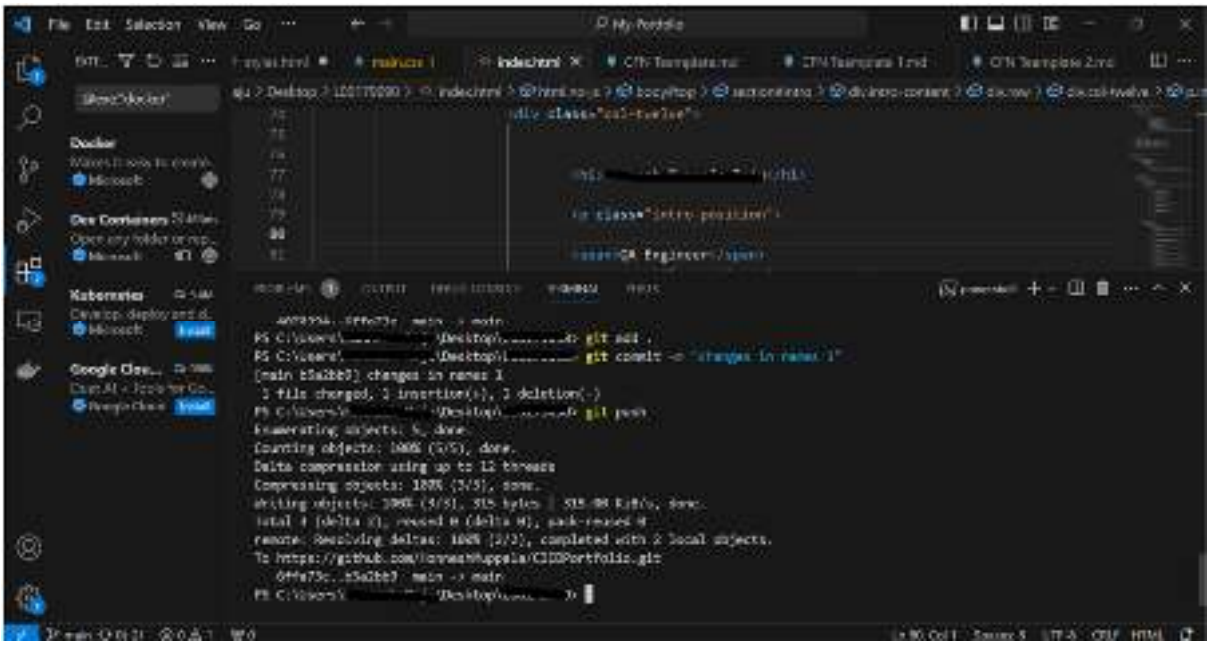


Webhook config

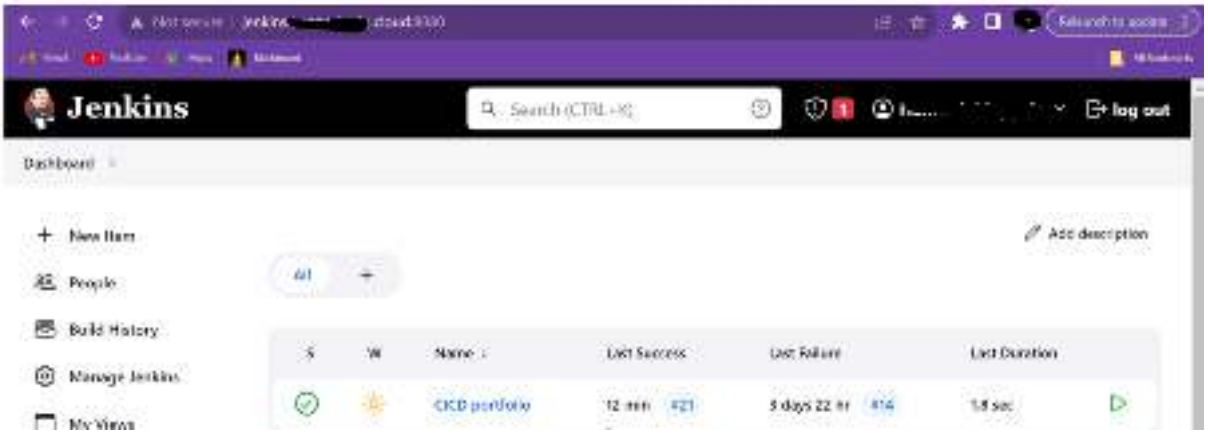


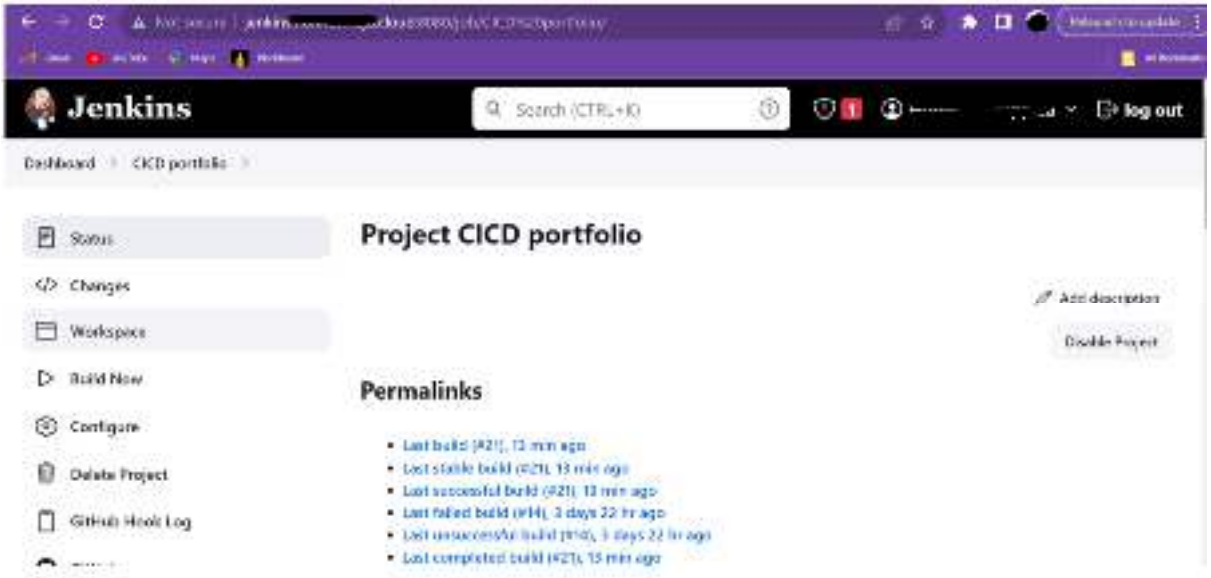


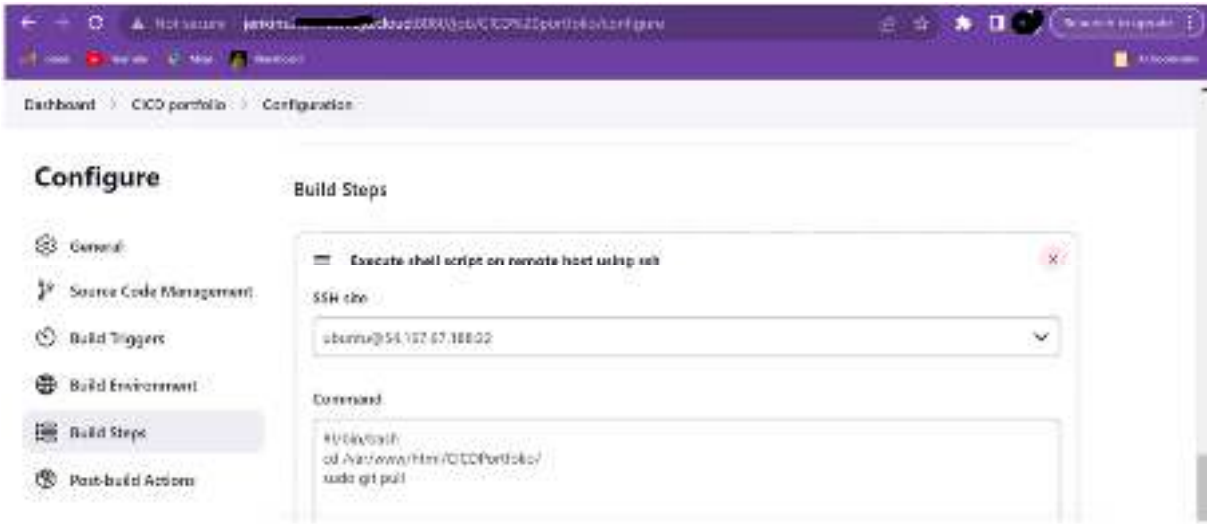
3. **VS code**

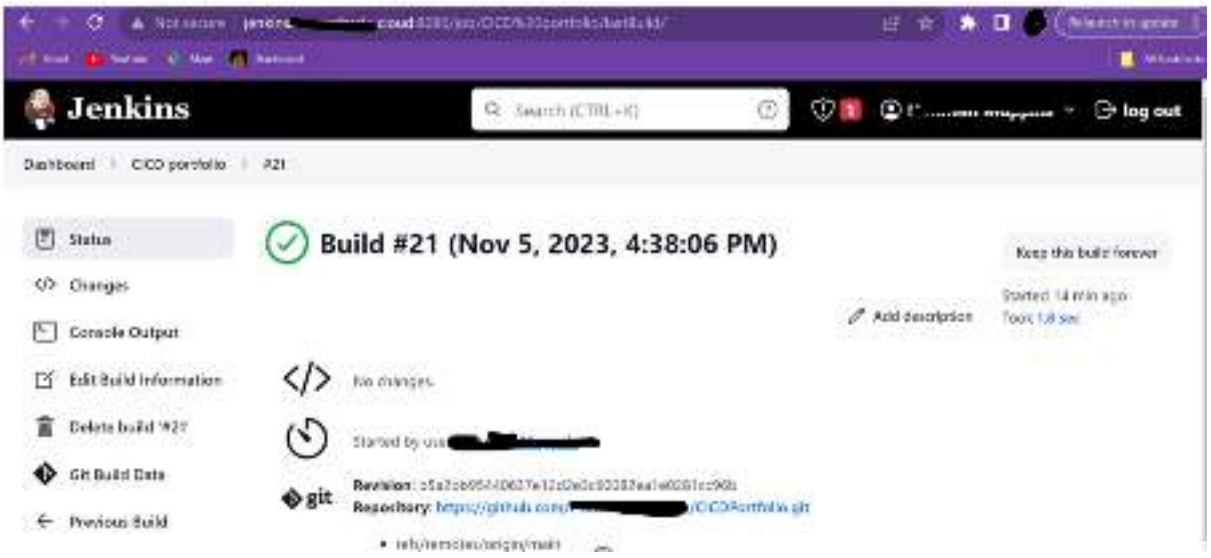


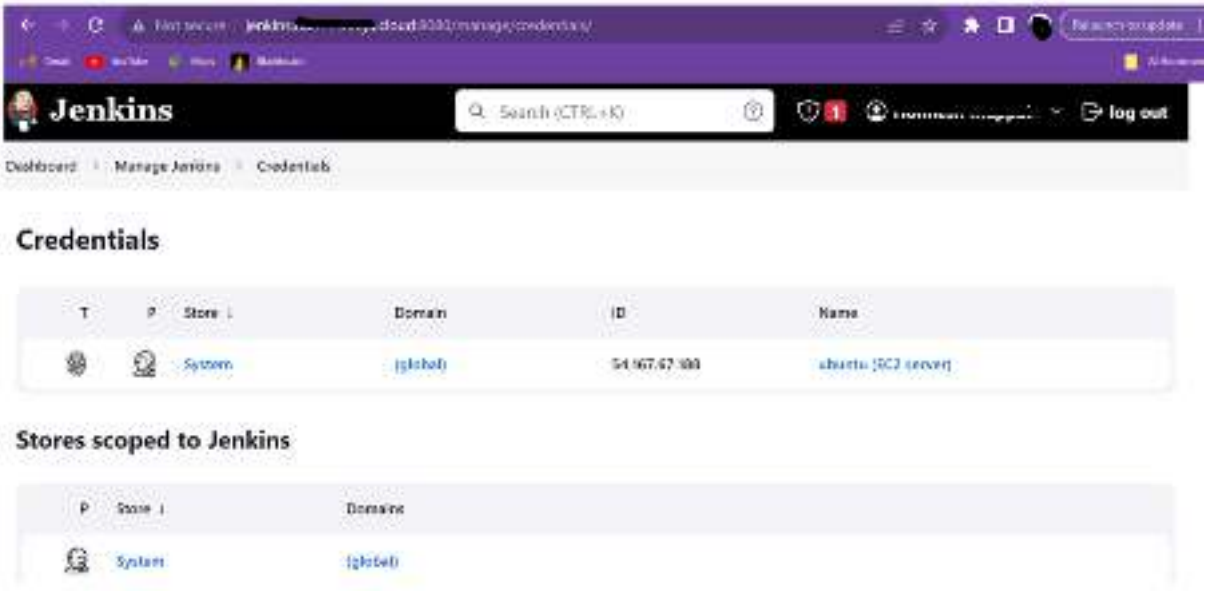
4. **Jenkins configurations and Settings**
   Link: http://jenkins.....

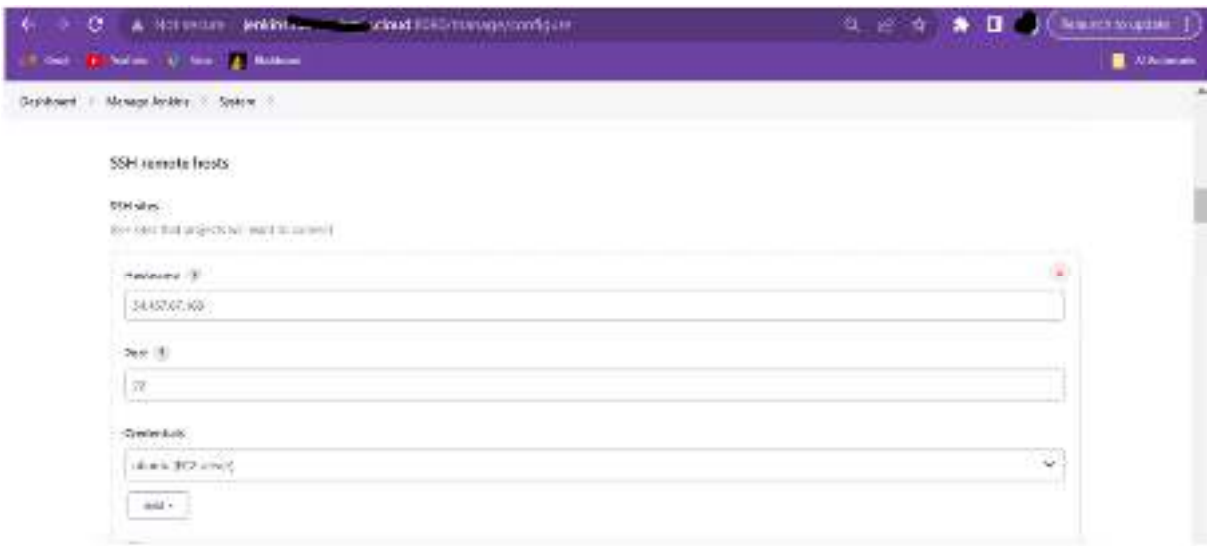

Jenkins
Search (CTRL+K)
log out
Dashboard
New Item
People
Build History
Manage Jenkins
My Views
Add description
All
S
W
Name
Last Success
Last Failure
Last Duration
CICD portfolio
12 min #21
5 days 22 hr #14
1.8 sec

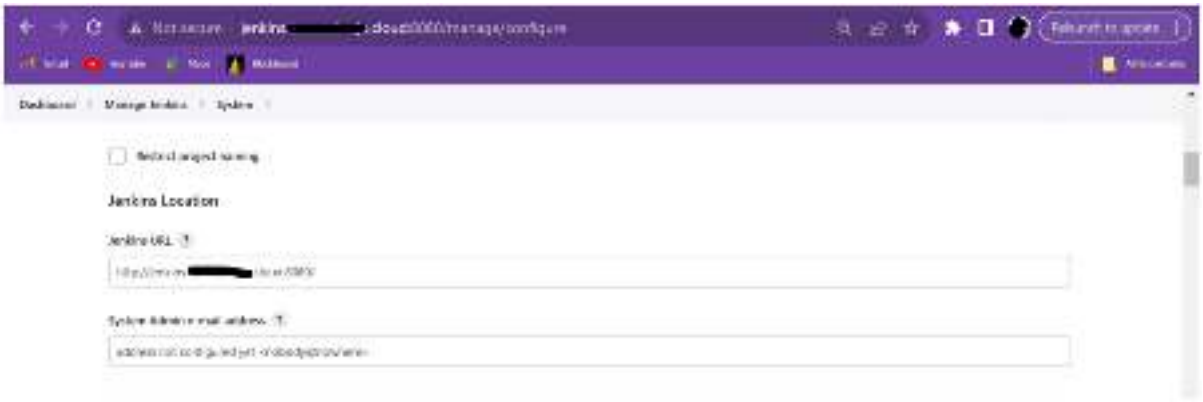

Jenkins
Search (CTRL+K)
log out
Dashboard
CICD portfolio
Status
Changes
Workspace
Build Now
Configure
Delete Project
GitHub Hook Log
Project CICD portfolio
Add description
Disable Project
Permalinks


Dashboard
CICD portfolio
Configuration
Configure
General
Source Code Management
Build Triggers
Build Environment
Build Steps
Post-build Actions
Build Steps
Execute shell script on remote host using ssh
SSH site
Command
cd /var/www/html/CICDPortfolio/
sudo git pull

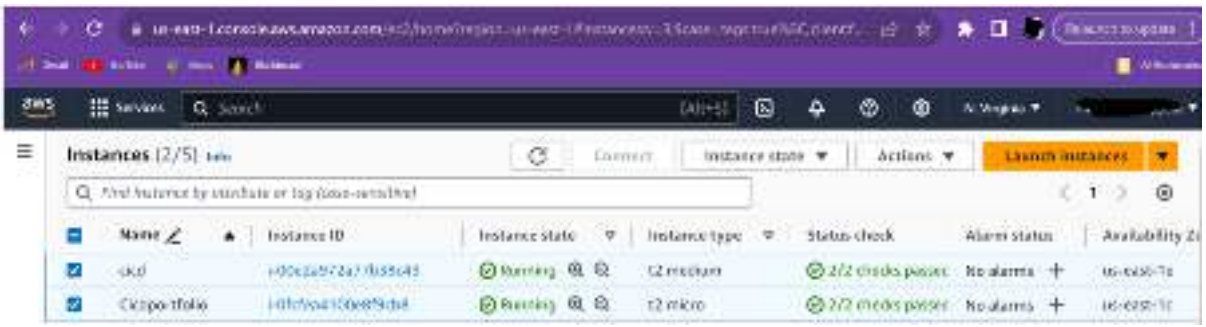

Jenkins
Search (CTRL+K)
log out
Dashboard
CICD portfolio
#21
Status
Changes
Console Output
Edit Build Information
Delete build '#21'
Git Build Data
Previous Build
Build #21 (Nov 5, 2023, 4:38:06 PM)
Keep this build forever
Add description
No changes.
git


Jenkins
Search (CTRL+K)
log out
Dashboard
Manage Jenkins
Credentials
Credentials
T
P
Store
Domain
ID
Name
System
(global)
ubuntu (EC2 server)
Stores scoped to Jenkins
P
Store
Domains
System
(global)


SSH remote hosts

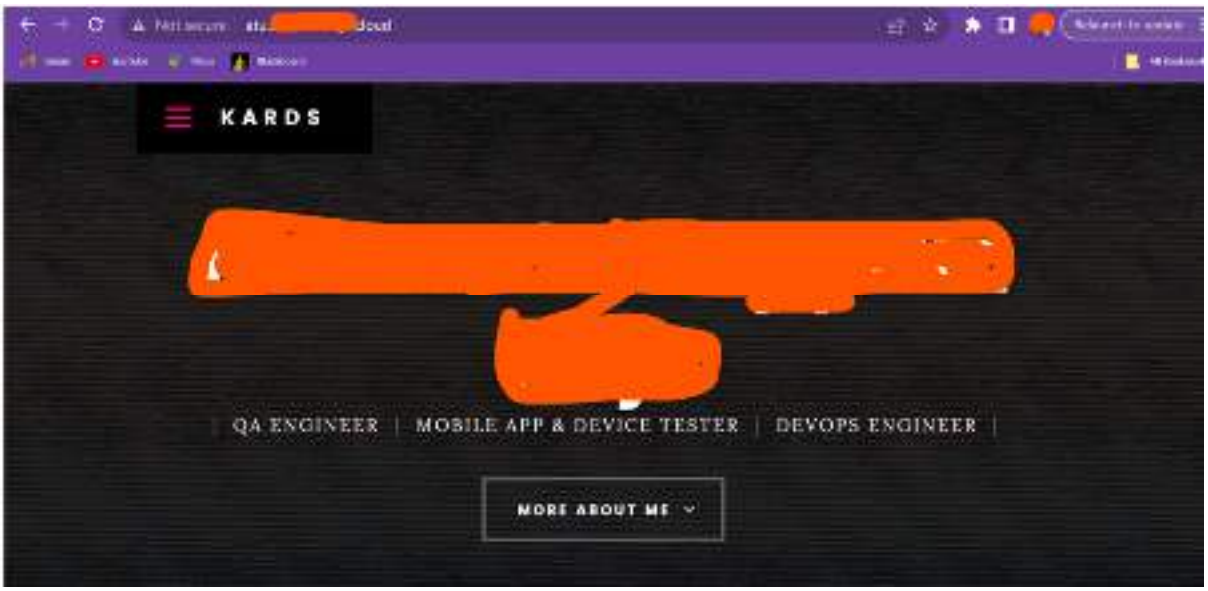


EC2 Instances



Portfolio

Link: http://atu.....



Conclusion:

In conclusion, the CI/CD pipeline I established using GitHub, EC2, and Jenkins played a pivotal role in streamlining the development, testing, and deployment processes for my Portfolio project. GitHub served as the source code repository, Jenkins automated the build and testing processes, and AWS EC2 instances were utilized for deployment, creating a robust and efficient pipeline. This approach ensured that the development and deployment of my project were smooth and efficient, ultimately contributing to the success of the Portfolio project. This section provides a detailed conclusion, summarizing the key benefits, outcomes, and the broader impact of the CI/CD pipeline on the project's success.

1. Streamlined Development Workflow:

The CI/CD pipeline effectively streamlined the development workflow. The integration of GitHub as the source code repository allowed for version control, collaboration, and efficient code management. Developers could work on features and fixes in isolated branches, ensuring that the main codebase remained stable. This led to a more organized and agile development process, with the assurance that code changes would be merged into the main branch only when they were thoroughly tested and reviewed.

2. Automated Build and Testing:

Jenkins, as the central automation tool, brought immense efficiency to the project. Automated builds reduced the manual effort required for compiling and packaging the application. Continuous integration enabled the team to detect and address issues early in the development cycle, leading to improved code quality. Automated testing ensured that each code change underwent rigorous evaluation, reducing the risk of introducing bugs or regressions into the project.

3. Reliable and Consistent Deployments:

The use of AWS EC2 instances for deployment to both staging and production environments ensured that deployments were reliable and consistent. The separation of staging and production environments on distinct EC2 instances minimized the risk of accidental changes affecting the live system. The Jenkins deployment process, triggered only after successful builds and tests, further enhanced the reliability of deployments.

4. Security and Access Control:

The CI/CD pipeline incorporated security best practices. GitHub's security features, including branch protection and access control, ensured that only authorized personnel could modify the codebase. Security groups in AWS EC2 instances limited network access, reducing the potential for security breaches. These measures collectively safeguarded the project's code and infrastructure.

5. Improved Collaboration:

The CI/CD pipeline facilitated collaboration among project team members and stakeholders. GitHub's collaborative features, such as pull requests and discussions, enabled effective code review and communication. The automation of tasks in Jenkins reduced manual coordination, allowing team members to focus on higher-value activities.

6. Continuous Integration and Continuous Delivery:

CI ensures that code changes are frequently and automatically integrated, providing a constant feedback loop to developers. CD allows for the automated delivery of changes to staging and, eventually, production environments. This approach reduces the time and effort required to release new features or fixes to end-users.

7. Impact on the Portfolio Project's Success:

The CI/CD pipeline has been a cornerstone of the Portfolio project's success. By automating key development, testing, and deployment processes, it has significantly improved the project's overall efficiency and reliability. It has allowed the project team to iterate and release code faster, reducing time-to-market and enhancing the end-user experience.

8. Future Scalability and Continuous Improvement:

The CI/CD pipeline isn't just a success for the current project but also lays the foundation for future endeavours. It enables easy scalability as the project grows, with the ability to add more features, team members, and environments. The insights gained from the implementation can be used to continuously improve the pipeline, ensuring that it remains an asset to the organization's software development efforts.

Acknowledgments:

I would like to express my gratitude to **Maria griffin**, my academic advisor, for continuous guidance and support throughout the development of this CI/CD pipeline. Additionally, I would like to thank the open-source communities behind GitHub, Jenkins, and AWS for providing invaluable tools and resources.

Bibliography

Article, D., 2021. How to install Jenkins on AWS Ec2 Ubuntu 20.04. *install Jenkins on AWS Ec2 Ubuntu 20.04.*

Attri, A., 2020. Configuring SSH connection to a remote host in Jenkins (SSH-plugin). *Configuring SSH connection.*

GitHub, n.d. GitHub Docs.

jenkins, n.d. Jenkins User Documentation. *Jenkins Documentation .*

Jr., G. B., 2023. Install Jenkins on AWS EC2 Instance(Ubuntu). *Install Jenkins on AWS EC2 Instance.*

services, A. w., n.d. Amazon Elastic Compute Cloud Documentation. *Amazon Elastic Compute Cloud Documentation.*