

## Title: Jenkins CI/CD with GitHub integration

---



## INDEX

<b>Sr.No.</b>	<b>Title</b>	<b>Page No.</b>
	<b>Introduction</b>	<b>3-4</b>
<b>1.</b>	<b>Launch an EC2 instance and configure jenkins on it.</b>  1.1] Create VPC for project 1.2] Create EC2 instance 1.3] Setup Jenkins and connect 1.4] Update security group 1.5] Connect Jenkins	<b>5-9</b>
<b>2.</b>	<b>Create new project on Jenkins</b>  2.1] Configure Jenkins with GitHub 2.2] Install Application Dependencies & Run Application 2.3] Update Dockerfile 2.4] Add Build Steps 2.5] Install Plugins & Setup Webhooks	<b>10- 20</b>
<b>3.</b>	<b>Change Code &amp; Verify Output</b> <b>Conclusion</b>	<b>21-22</b>

## ❖ Project Summary:

This project involved creating a Continuous Integration and Continuous Deployment (CI/CD) pipeline for a Node.js Todo List application. I set up an AWS EC2 instance to host Jenkins, which automates the process of building, testing, and deploying the application.

Using Docker, I created a container that includes all necessary dependencies, ensuring consistent performance across environments. The pipeline is configured to pull code from a GitHub repository, and with GitHub webhooks, it automatically triggers the build and deployment process whenever changes are made.

This setup enhances development efficiency by reducing manual work, providing quick feedback on code updates, and allowing for easy scaling of the application. Overall, the project showcases my DevOps skills and my ability to implement modern automation practices for software development.

## ❖ Tools and Technologies Used

- **Jenkins:** An open-source automation server used for building, testing, and deploying applications. It serves as the backbone of our CI/CD pipeline.
- **Docker:** A platform that allows developers to automate the deployment of applications inside lightweight, portable containers. This ensures that the application runs consistently across different environments.
- **AWS EC2:** Amazon Web Services Elastic Compute Cloud (EC2) was utilized to host the Jenkins server. This cloud service provides scalable computing capacity in the AWS cloud.
- **GitHub:** A version control platform that hosts the application code. GitHub was integrated with Jenkins to facilitate automatic code deployment upon changes.
- **Java:** Required for running Jenkins, as it is built on Java technology.

## ❖ CI/CD Pipeline Configuration

- **Jenkins Job Creation:** I created a new job in Jenkins using the Freestyle project type. This job was configured to pull the latest code from the GitHub repository whenever changes were made.
- **SSH Key Integration:** To ensure secure communication between Jenkins and the EC2 instance, I set up SSH key credentials. This enabled Jenkins to execute commands on the EC2 instance without manual authentication.
- **Build Steps:** In the Jenkins job configuration, I added build steps that included:
  1. **Docker Image Creation:** Jenkins was set up to build a Docker image that contains all the necessary dependencies for the Node.js Todo List application. This image encapsulates the application code and its environment, making it portable and easy to deploy.
  2. **Container Deployment:** After successfully building the Docker image, Jenkins automatically deploys it as a container, ensuring that the application is running and accessible.

### ❖ Benefits Achieved

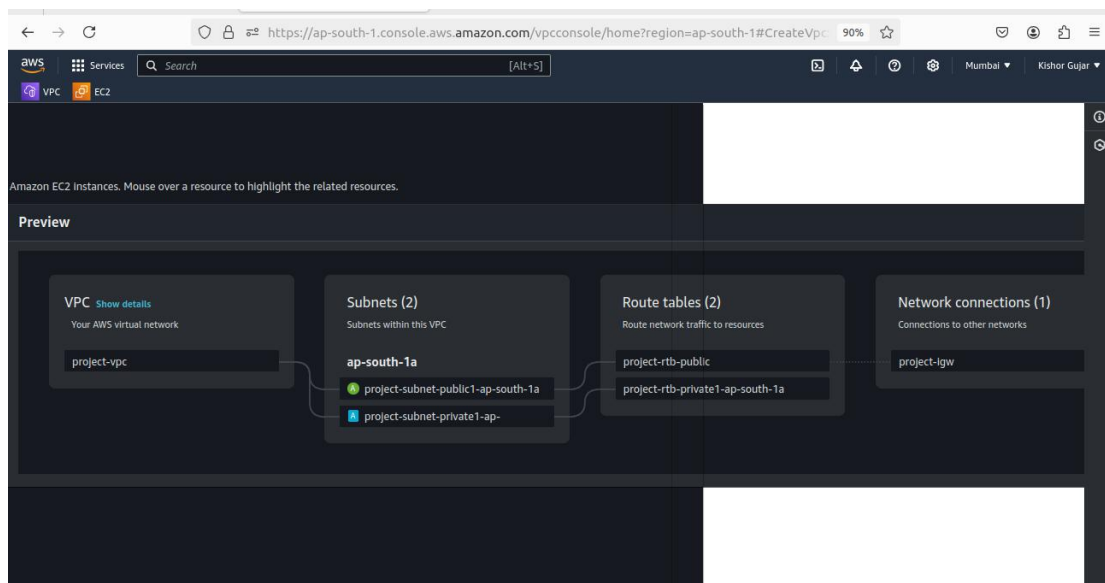
- **Automation:** The CI/CD pipeline automates the entire process from code commit to deployment, significantly reducing manual effort and the potential for human error.
- **Rapid Feedback:** Automated testing and deployment provide quick feedback on code changes, allowing for faster iterations and improvements.
- **Scalability:** Using Docker allows for easy scaling of the application, as containers can be quickly replicated to handle increased traffic.

**Let's Play It On Ground.....**

## 1] Launch an EC2 instance and configure jenkins on it.

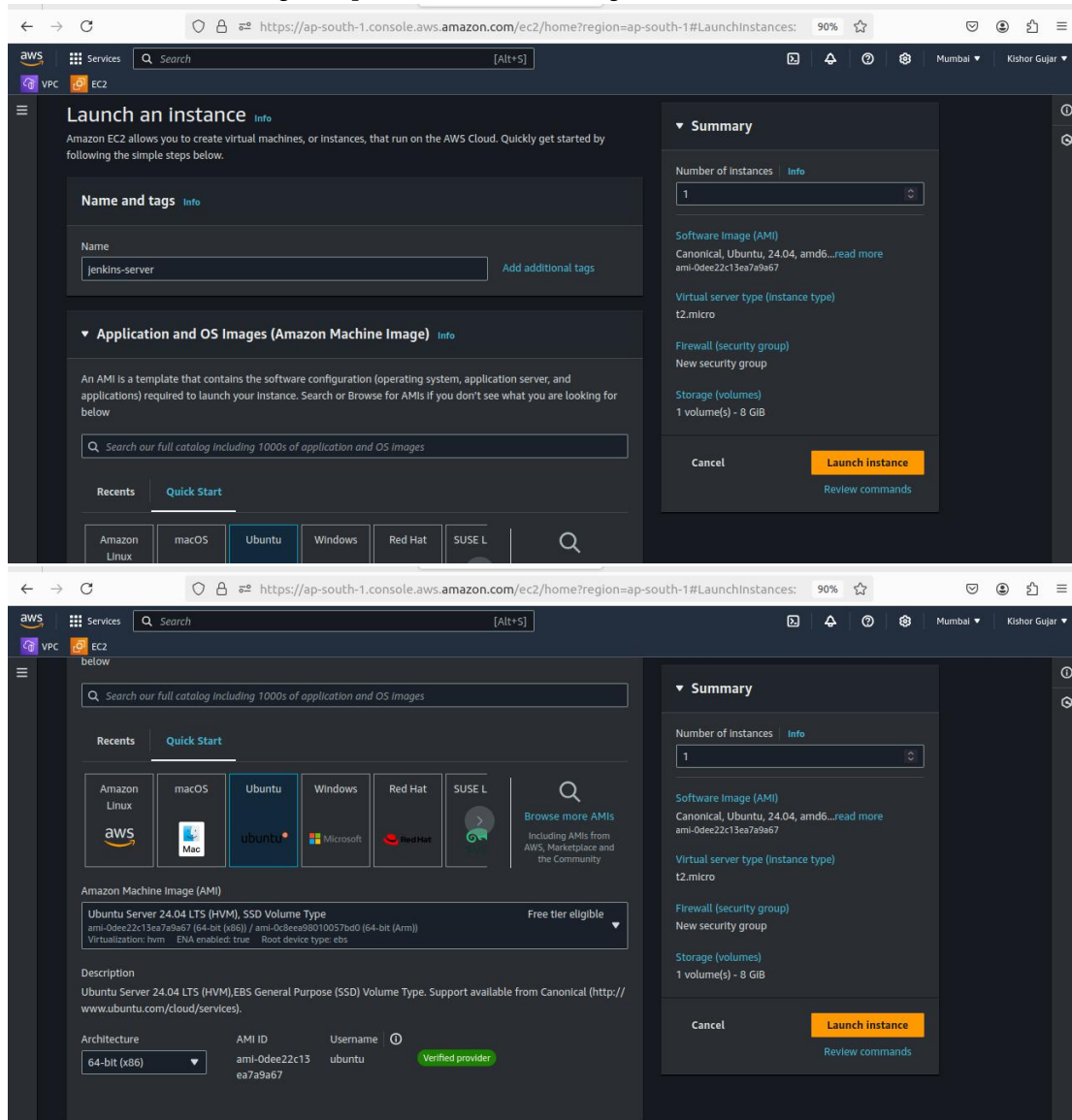
### 1.1] Create VPC for project

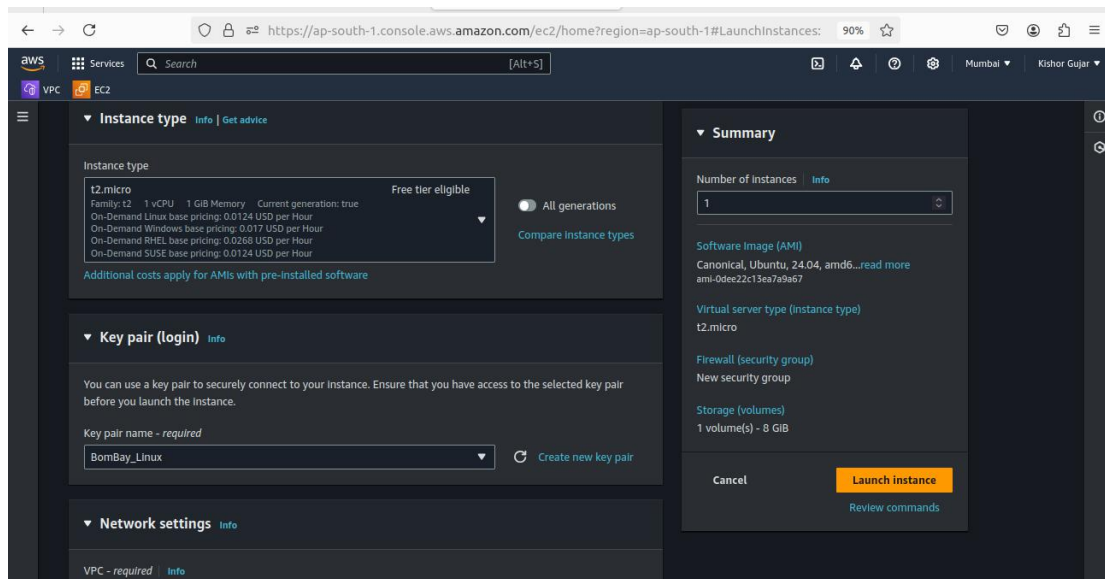
Creating a Virtual Private Cloud (VPC) allows for network isolation, enhanced security, and control over your AWS infrastructure. To set up a VPC, navigate to the VPC dashboard, create a new VPC with a specified CIDR block, and then create one public and one private subnet within the same availability zone. Attach an Internet Gateway (IGW) to the VPC and update the public subnet's route table to allow outbound traffic to the internet. Finally, configure security groups to manage traffic. This setup enables a secure environment for your applications.



## 1.2] Create EC2 instance

To create an Ubuntu EC2 instance, log in to the AWS Management Console and navigate to the EC2 dashboard. Click on Launch Instance, select an Ubuntu AMI, and choose the t2.micro instance type. In the configuration settings, select your VPC and public subnet, and ensure "Auto-assign Public IP" is enabled. Use the default security group, allowing inbound SSH traffic, and select your key pair for access. Finally, review your settings and launch the instance. You can connect to it via SSH using the public IP address assigned to the instance.





### 1.3] Setup Jenkins and connect

Jenkins is an open-source automation server that facilitates continuous integration and continuous deployment (CI/CD) of software projects. I chose Jenkins for this project because it automates the building, testing, and deployment processes, thereby enhancing development efficiency and allowing for quick feedback on code changes.

#### Commands to Set Up Jenkins

```
# sudo su
```

(Switch to the root user for administrative privileges.)

```
# sudo apt-get update -y
```

(Update the package list to ensure all available packages are up-to-date.)

```
# sudo apt install fontconfig openjdk-17-jre
```

(Install the fontconfig and OpenJDK 17 Java Runtime Environment, which is required for Jenkins.)

```
# java -version
```

(Verify the installation of Java and check the installed version.)

```
# sudo wget -O /usr/share/keyrings/jenkins-keyring.asc
```

```
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
```

(Download the Jenkins repository key to ensure package authenticity.)

```
# echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]
```

```
https://pkg.jenkins.io/debian-stable binary/" | sudo tee
```

```
/etc/apt/sources.list.d/jenkins.list > /dev/null
```

(Add the Jenkins repository to the system's package sources list.)

**# sudo apt-get update**

(Update the package list again to include the newly added Jenkins repository.)

**# sudo apt-get install jenkins**

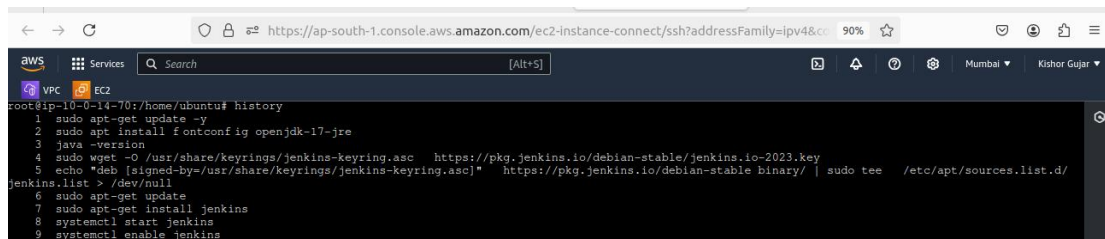
(Install Jenkins from the added repository.)

**# systemctl start jenkins**

(Start the Jenkins service to begin using it.)

**# systemctl enable jenkins**

(Enable Jenkins to start automatically at system boot.)

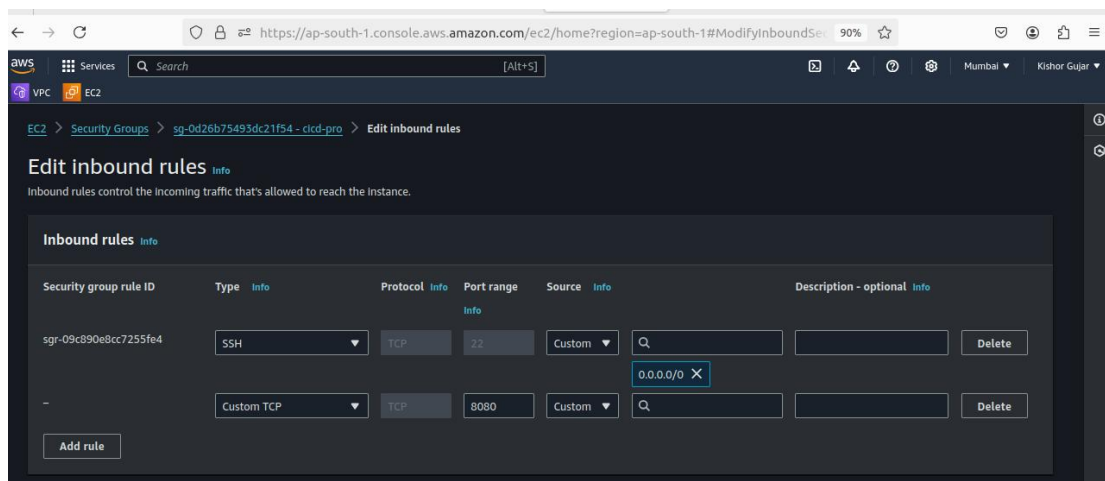


```
root@ip-10-0-14-70:/home/ubuntu# history
1 sudo apt-get update -y
2 sudo apt install fontconfig openjdk-17-jre
3 java -version
4 sudo wget -O /usr/share/keyrings/jenkins-keyring.asc https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
5 echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/" | sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null
6 sudo apt-get update
7 sudo apt-get install jenkins
8 systemctl start jenkins
9 systemctl enable jenkins
```

#### 1.4] Update security group

A security group acts as a virtual firewall for your AWS resources, controlling inbound and outbound traffic. **Inbound rules** specify which incoming connections are allowed, while **outbound rules** determine what outgoing traffic is permitted.

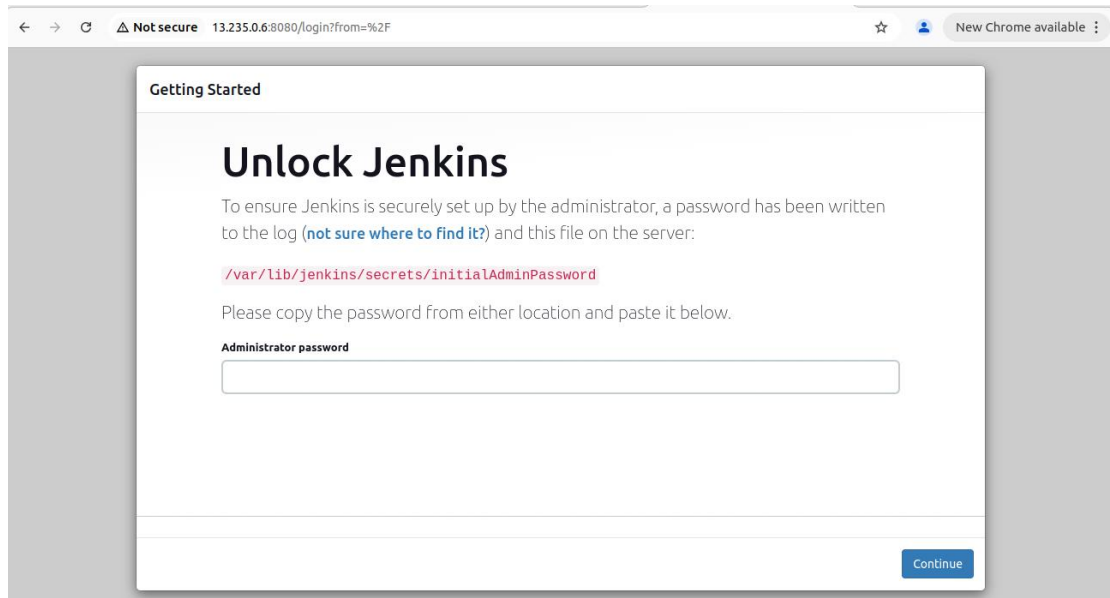
We need to open port **8080** in the inbound rules to allow external access to the Jenkins web interface, enabling users to interact with Jenkins for managing builds and deployments. This is essential for the CI/CD pipeline to function effectively, as it allows developers to monitor and control the automated processes.





### 1.5] Connect Jenkins

To connect to Jenkins, copy the public IP address of your AWS EC2 instance. In your web browser, enter the address as **http://<your-public-ip>:8080**. This will take you to the Jenkins login page, where you can access the Jenkins interface and manage your CI/CD pipeline.



To access Jenkins for the first time, you need the default admin password, which is stored in the file `/var/lib/jenkins/secrets/initialAdminPassword`. This password allows you to unlock Jenkins and proceed with the initial setup. Use the following command to retrieve it:

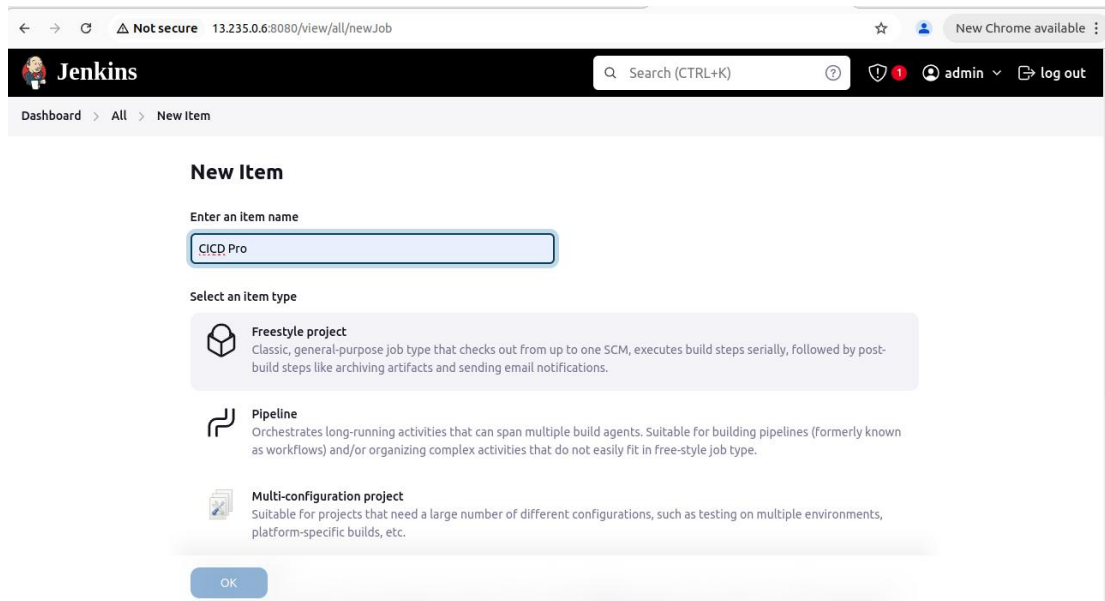
```
# sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

(Displays the default admin password required to unlock Jenkins.)

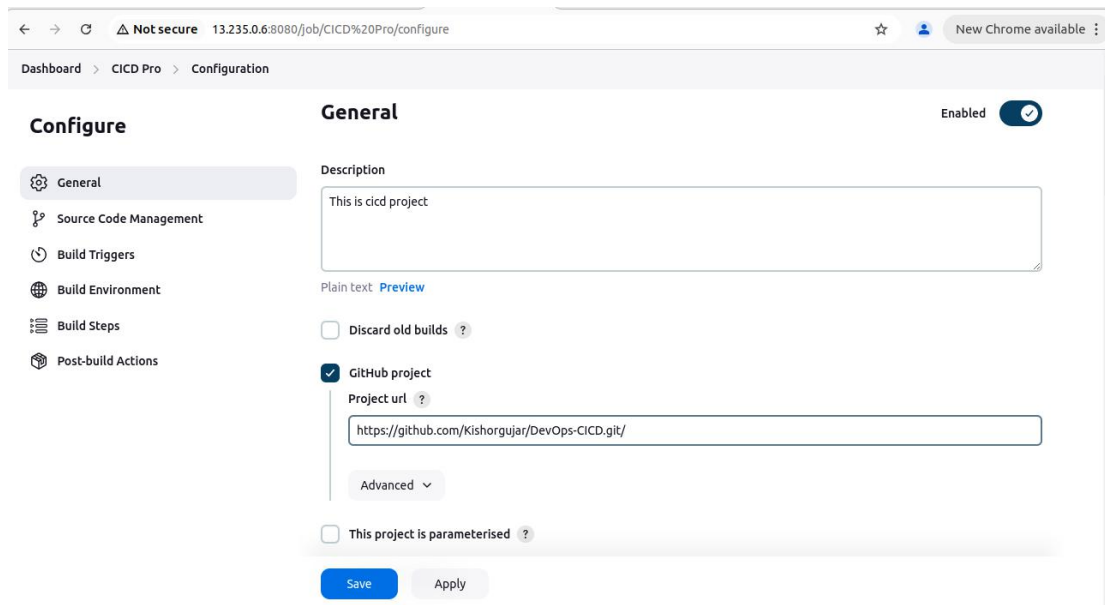
## 2] Create new project on Jenkins

### 2.1] Configure Jenkins with GitHub

After retrieving the default admin password, access the Jenkins web interface to set up your username and password, and install the recommended default plugins for initial functionality. Once the setup is complete, navigate to the Jenkins **dashboard**, click on "New Item," select "**Freestyle project**," and click "OK" to proceed with creating a new project. This allows you to configure and automate your build processes.



To link your Jenkins project with GitHub, open your terminal and run the command `git clone https://github.com/Kishorgujar/DevOps-CICD` to download the project files to your local machine. After cloning, go to the Jenkins dashboard and open your freestyle project. In the "Source Code Management" section, paste the GitHub repository link. This will enable Jenkins to automatically pull the latest code from GitHub for continuous integration and deployment.



Dashboard > CICD Pro > Configuration

## Configure

General **Enabled** ☒

- General
- Source Code Management
- Build Triggers
- Build Environment
- Build Steps
- Post-build Actions

**General**

Description  
This is cicd project

Plain text [Preview](#)

☐ Discard old builds ?

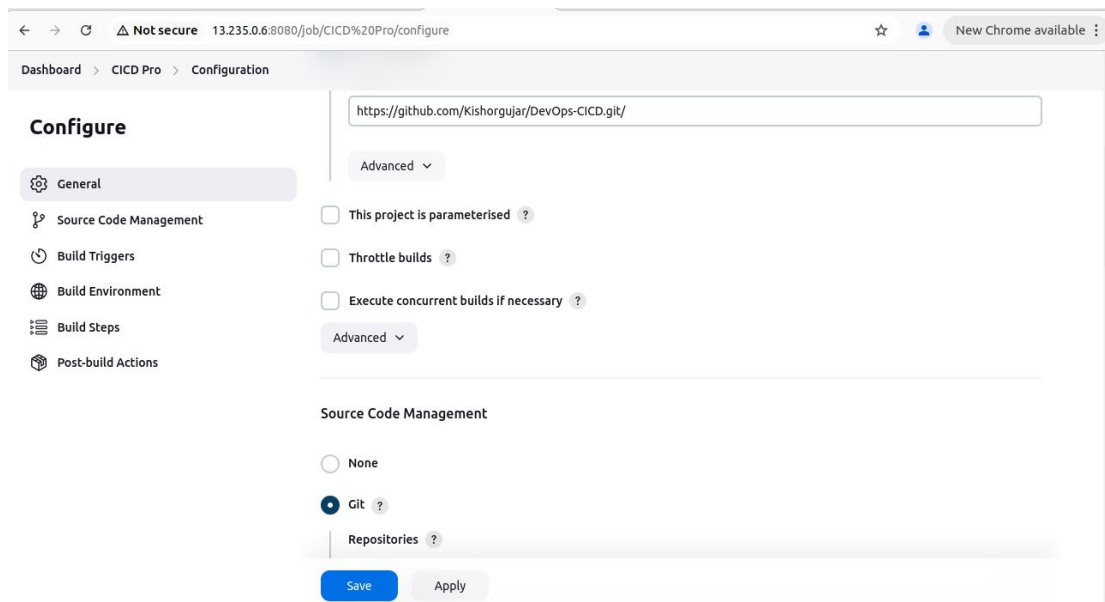
☒ GitHub project

Project url ?  
<https://github.com/Kishorgujar/DevOps-CICD.git/>

Advanced ▾

☐ This project is parameterised ?

[Save](#) [Apply](#)



Dashboard > CICD Pro > Configuration

## Configure

General

- General
- Source Code Management
- Build Triggers
- Build Environment
- Build Steps
- Post-build Actions

<https://github.com/Kishorgujar/DevOps-CICD.git/>

Advanced ▾

☐ This project is parameterised ?

☐ Throttle builds ?

☐ Execute concurrent builds if necessary ?

Advanced ▾

**Source Code Management**

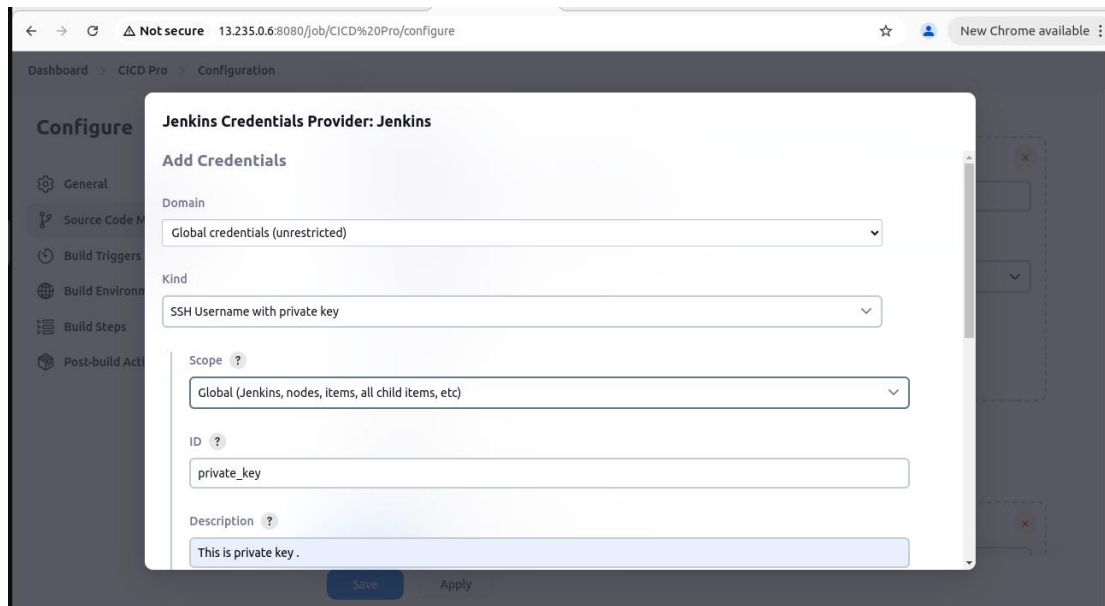
☐ None

☒ Git ?

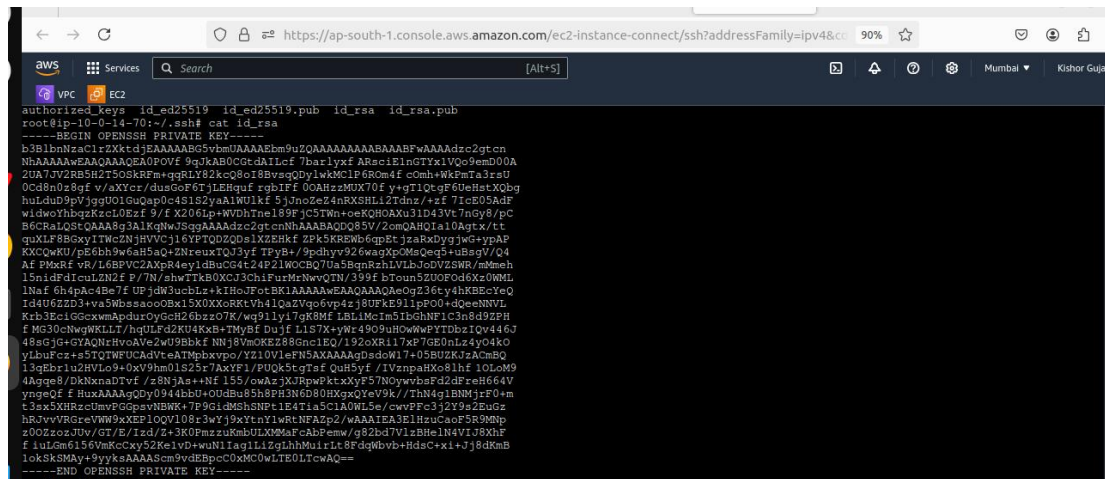
Repositories ?

[Save](#) [Apply](#)

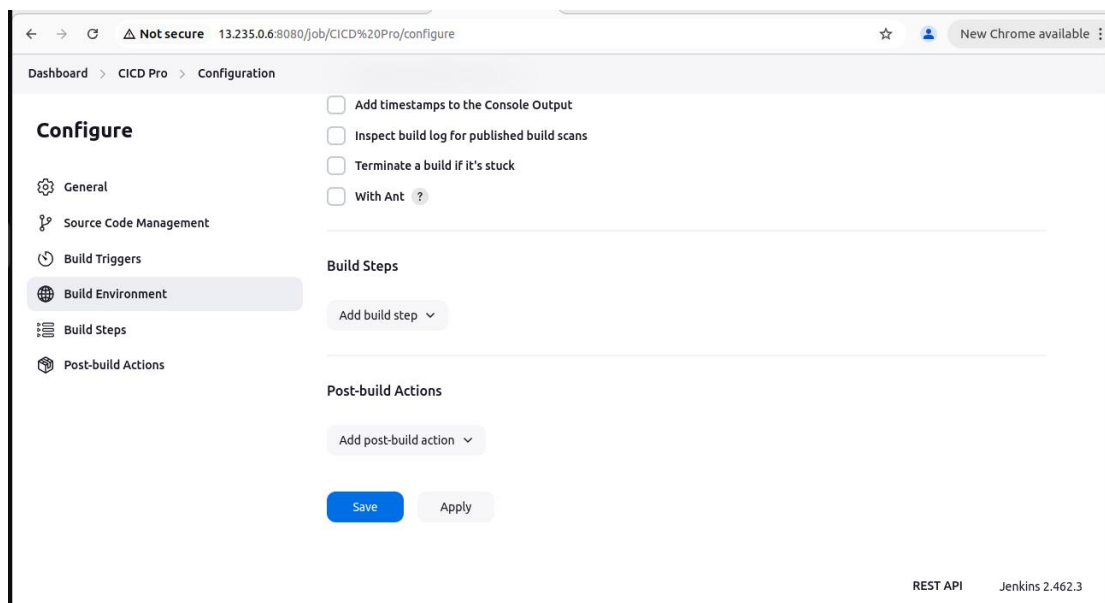
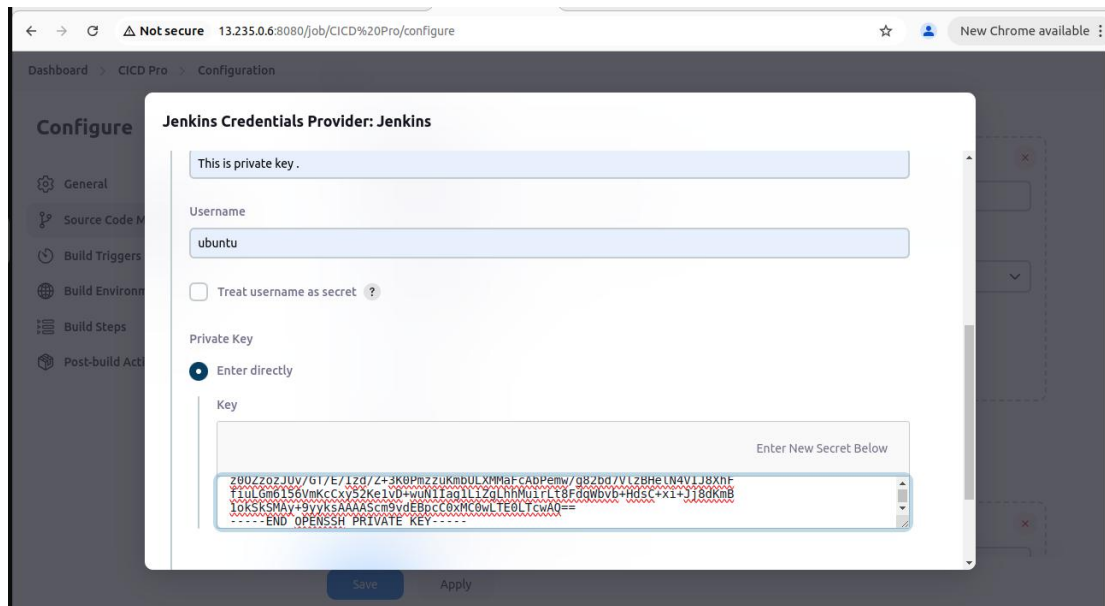
After selecting Git as the source code management option in Jenkins, you'll need to provide credentials to enable Jenkins to operate on your EC2 instance. Click "Add" next to the Credentials field and choose "SSH Username with private key." Enter an ID and description of your choice, and for the username, use the one you logged in with— in my case, it's "ubuntu." This setup allows Jenkins to securely connect to your EC2 instance for executing build and deployment tasks.



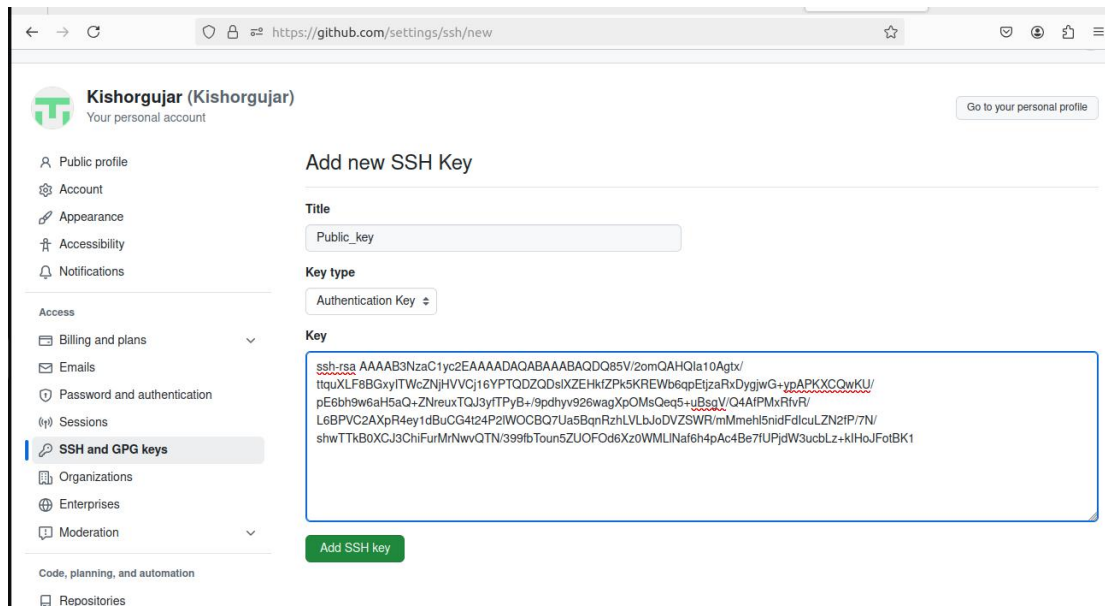
To generate an SSH key for Jenkins, use the command `ssh-keygen` and simply press Enter twice to accept the default settings. Next, navigate to the directory by running `cd /root/.ssh`, where you'll find your SSH keys. The private key is usually named `id_rsa`, and the public key is `id_rsa.pub`. To provide Jenkins with the private key, execute `cat id_rsa` to display it, and then copy the entire key. This private key will be used to authenticate Jenkins when connecting to your EC2 instance.



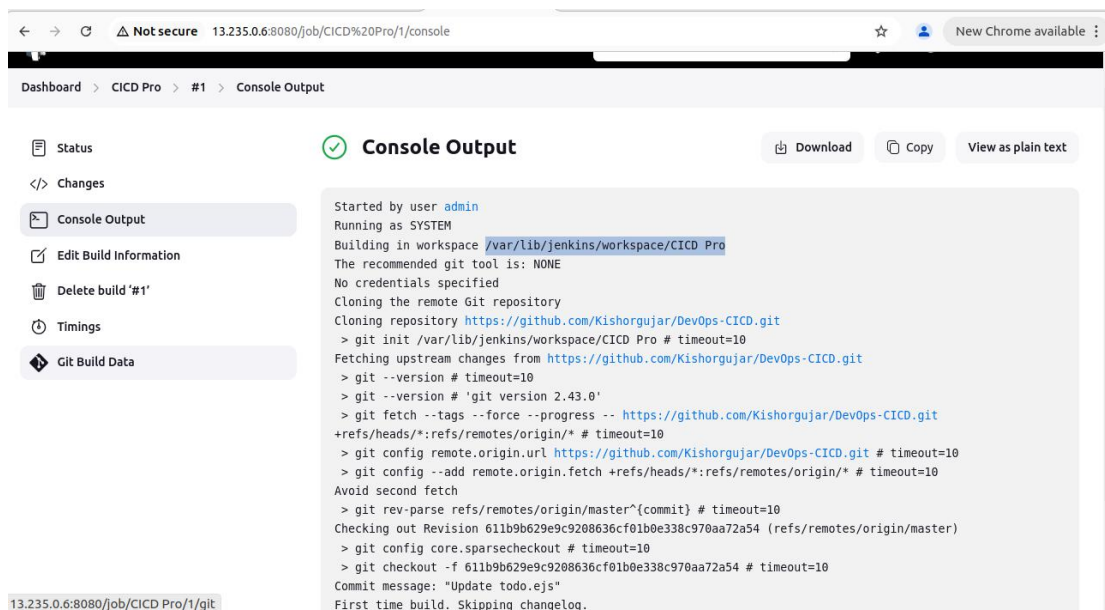
After copying your SSH private key, return to the Jenkins page. In the credentials section, select "Add" and choose "SSH Username with private key." In the key column, paste your SSH private key, then click "Save" to store the credentials. This allows Jenkins to securely connect to your EC2 instance.

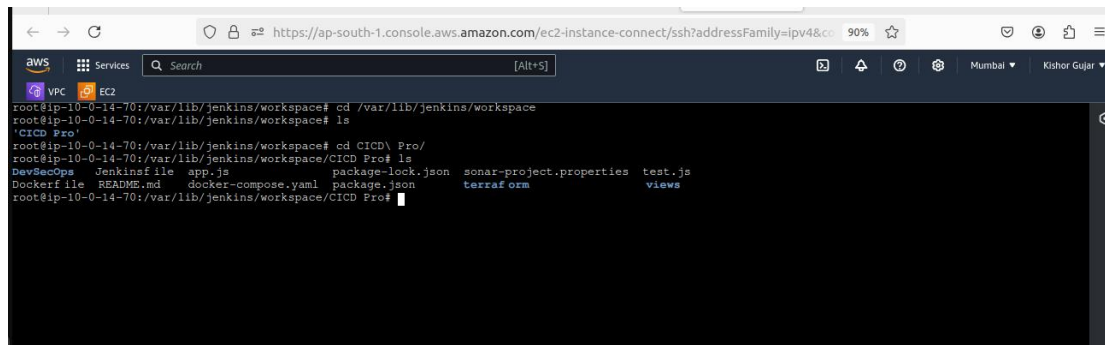


Next, go to your GitHub account and navigate to “**Settings**”. Select “**SSH and GPG keys**” from the sidebar. Click on “**New SSH key**”, give it a title of your choice, and select “**Authentication type**” as the key type. Paste your public SSH key (copied from the `~/.ssh` directory on your instance) into the provided field. Finally, click “**Add SSH key**” to save it, enabling your EC2 instance (where Jenkins is running) to securely access your GitHub repository.



Now, return to your Jenkins web page and click on Build Now to start the build process. Once the build is successful, click on the build number (e.g., “#1”) and then select “**Console Output**” from the right sidebar. Look for the line that says “**Building in workspace**” and copy the path shown. Next, go to your EC2 instance terminal and execute ``cd <paste path>`` to navigate to that directory. You should see all the files from your GitHub repository in that workspace.

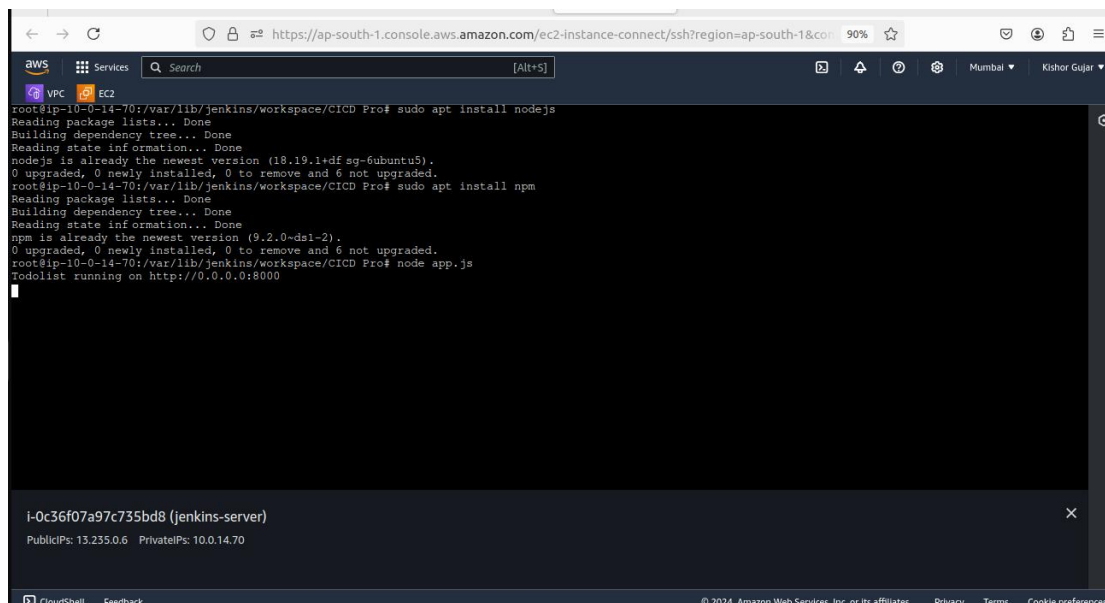




```
root@ip-10-0-14-70:/var/lib/jenkins/workspace# cd /var/lib/jenkins/workspace
root@ip-10-0-14-70:/var/lib/jenkins/workspace# ls
'CICD Pro'
root@ip-10-0-14-70:/var/lib/jenkins/workspace# cd CICD\ Pro/
root@ip-10-0-14-70:/var/lib/jenkins/workspace/CICD Pro# ls
DevSecOps  Jenkinsfile  app.js      package-lock.json  sonar-project.properties  test.js
Dockerfile README.md    docker-compose.yml  package.json        terraform               views
root@ip-10-0-14-70:/var/lib/jenkins/workspace/CICD Pro#
```

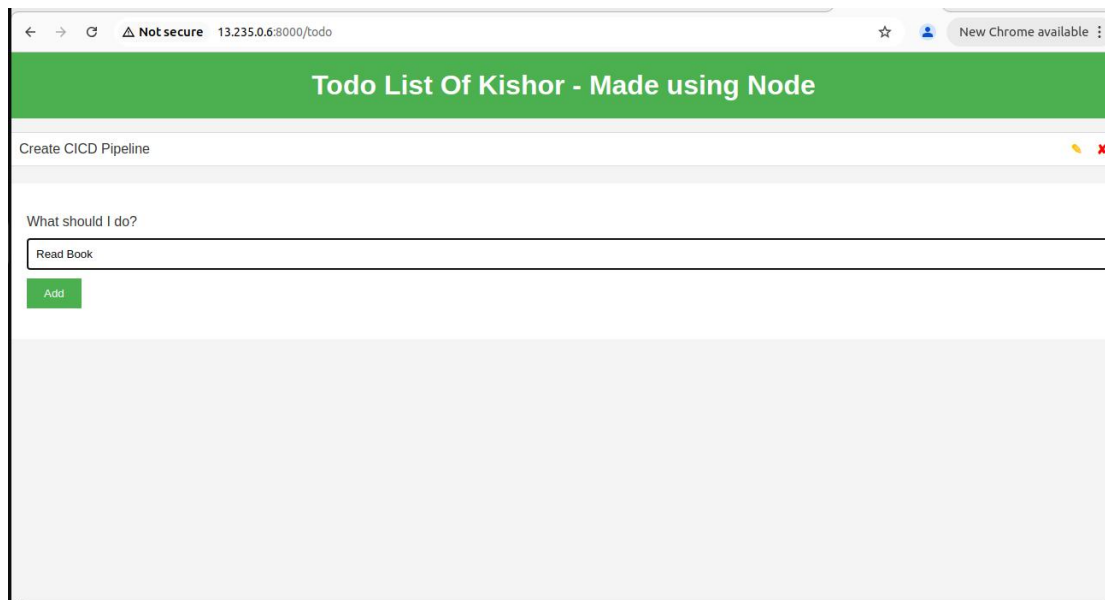
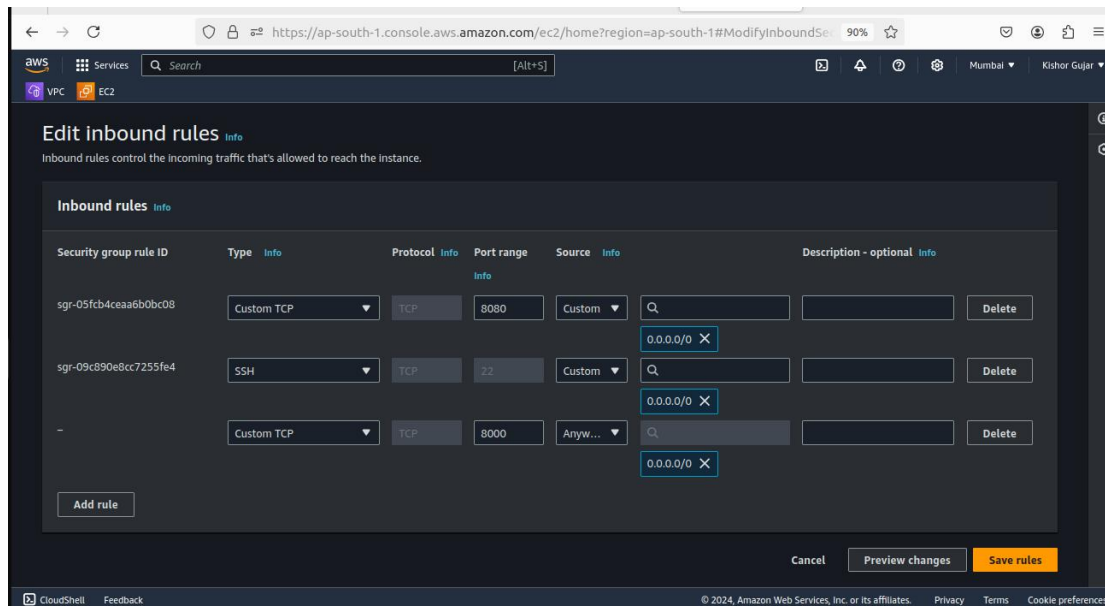
## 2.2] Install Application Dependencies & Run Application

To run your application, you'll need to install some dependencies listed in the `README.md` file. First, install Node.js by executing `sudo apt install nodejs` in your EC2 terminal. Next, install npm using `sudo apt install npm`. After the installations, start the application with the command `node app.js`. Since the application runs on port 8000, you must modify the security group settings. Go to your AWS Management Console, find the security group associated with your EC2 instance, and add an inbound rule to allow traffic on **port 8000**. Finally, open your web browser and enter your instance's public IP followed by `:8000` (e.g., `http://your-public-ip:8000`). If everything is set up correctly, you should see your application running!



```
root@ip-10-0-14-70:/var/lib/jenkins/workspace/CICD Pro# sudo apt install nodejs
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
nodejs is already the newest version (18.19.1+dfsg-6ubuntu5).
0 upgraded, 0 newly installed, 0 to remove and 6 not upgraded.
root@ip-10-0-14-70:/var/lib/jenkins/workspace/CICD Pro# sudo apt install npm
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
npm is already the newest version (9.2.0~ds1-2).
0 upgraded, 0 newly installed, 0 to remove and 6 not upgraded.
root@ip-10-0-14-70:/var/lib/jenkins/workspace/CICD Pro# node app.js
Todolist running on http://0.0.0.0:8000
```

i-0c36f07a97c735bd8 (jenkins-server)  
PublicIPs: 13.235.0.6 PrivateIPs: 10.0.14.70



## 2.3] Update Dockerfile

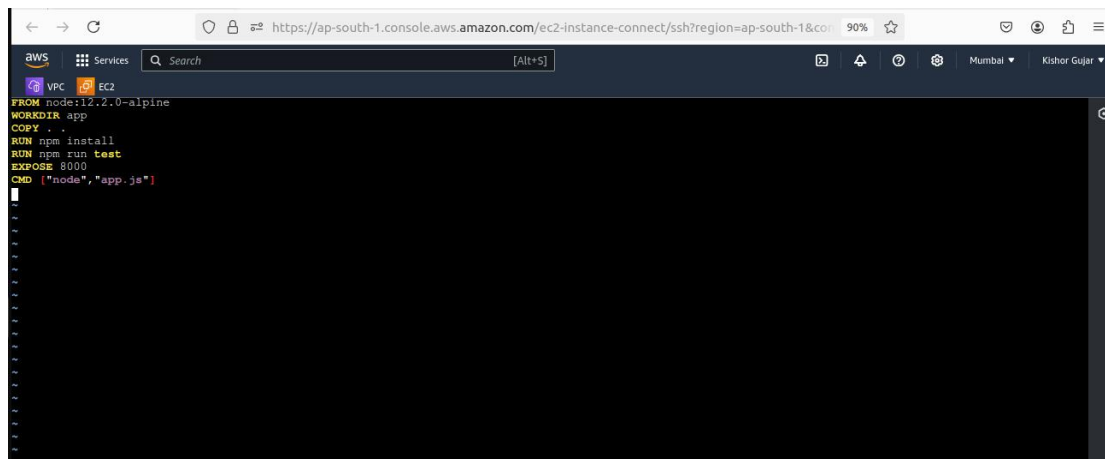
To keep your application running even after closing the terminal, you'll need to create a Dockerfile. First, delete the existing Dockerfile with the command **"rm -rf Dockerfile"**. Next, create a new Dockerfile by running **"vim Dockerfile"** and paste the following content:

```
FROM node:12.12.0-alpine #Use the Node.js Alpine image for a lightweight container
WORKDIR app              # Set the working directory inside the container
COPY . .                  # Copy all application files to the container
RUN npm install           # Install dependencies specified in package.json
EXPOSE 8000               # Expose port 8000 for the application
CMD ["npm", "start"]     # Command to start the application
```

After pasting the content, save the Dockerfile. This setup will allow your



application to run inside a Docker container, ensuring it stays active even when the terminal is closed.



```
FROM node:12.2.0-alpine
WORKDIR app
COPY . .
RUN npm install
RUN npm run test
EXPOSE 8000
CMD ["node", "app.js"]
```

## 2.4] Add Build Steps

Now, go to your Jenkins web page and click on your project, then select **config**. In the Build Steps section, choose the Execute shell option. Paste the following commands:

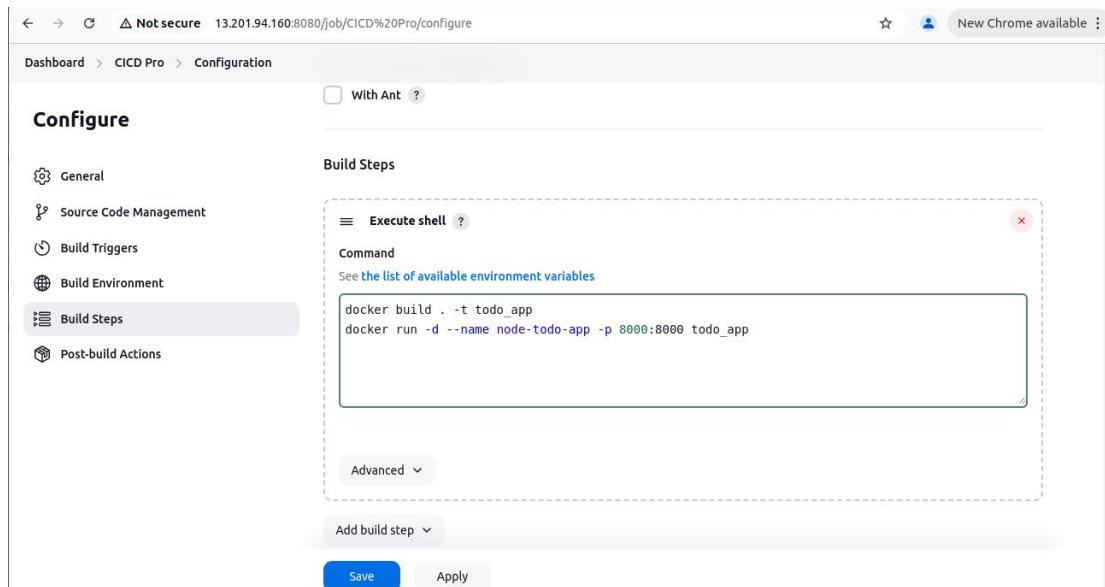
```
# docker build . -t todo_app
```

(Builds a Docker image named "todo\_app" using the current directory.)

```
#docker run -d --name node-todo-app -p 8000:8000 todo_app
```

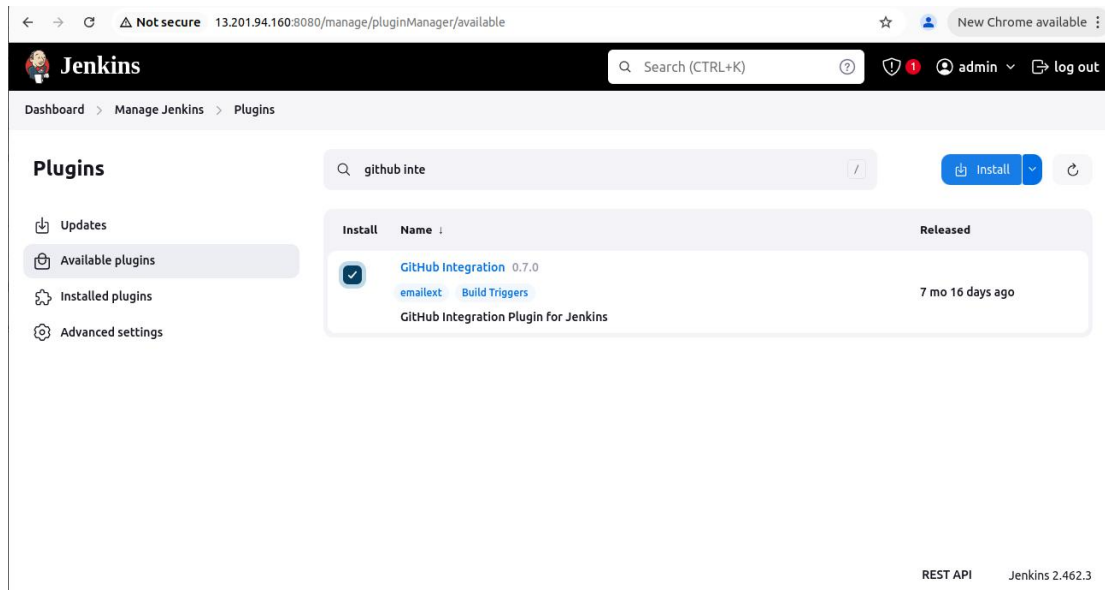
(Runs the "todo\_app" image in detached mode, mapping port 8000 of the container to port 8000 on the host.)

Finally, click the Save button to store your changes. This setup enables Jenkins to automate the continuous deployment process for your application using Docker.



## 2.5] Install Plugins & Setup Webhooks

To connect Jenkins with GitHub, you'll need to install the **“GitHub Integration”** plugin. Go to your Jenkins web page, click on **“Manage Jenkins”**, then select **“Manage Plugins”**. In the **“Available Plugins”** tab, use the search bar to find "GitHub Integration." Once you see the plugin, install it, and then click on **“Restart Jenkins”** to apply the changes. This integration will allow Jenkins to automatically trigger builds based on changes in your GitHub repository.



To complete the integration, go to your GitHub repository and click on **“Settings”**. In the right sidebar, select **“Webhooks”** and then click on **“Add webhook”**. In the **“Payload URL”** field, paste your Jenkins URL, which is your instance's **public IP** followed by **`:8080/github-webhook/`**. Set the **“Content type”** to **`application/json`** and leave the rest of the options as default. Finally, click on **“Add webhook”**. Ensure that you see a **green checkmark** next to the webhook to confirm it's set up correctly, indicating that GitHub can communicate with your Jenkins instance.

← → ↻ Kishorgujar / DevOps-CICD  + 🔍 📄 📧

<> Code Issues Pull requests Actions Projects Wiki Security Insights **Settings**

General

Access

- Collaborators
- Moderation options

Code and automation

- Branches
- Tags
- Rules
- Actions
- Webhooks**
- Environments
- Codespaces
- Pages

Security

- Code security

### Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

**Payload URL \***

**Content type \***

application/json

**Secret**

**SSL verification**

By default, we verify SSL certificates when delivering payloads.

☒ Enable SSL verification ☐ Disable (not recommended)

**Which events would you like to trigger this webhook?**

☐ Just the push event.

☒ Send me **everything**.

☐ Let me select individual events.

☒ **Active**

We will deliver event details when this hook is triggered.

Add webhook

← → ↻ Kishorgujar / DevOps-CICD  + 🔍 📄 📧

<> Code Issues Pull requests Actions Projects Wiki Security Insights **Settings**

Actions

**Webhooks**

- Environments
- Codespaces
- Pages

Security

- Code security
- Deploy keys
- Secrets and variables

Integrations

- GitHub Apps
- Email notifications

### Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

**Payload URL \***

**Content type \***

application/json

**Secret**

**SSL verification**

By default, we verify SSL certificates when delivering payloads.

☒ Enable SSL verification ☐ Disable (not recommended)

**Which events would you like to trigger this webhook?**

☐ Just the push event.

☒ Send me **everything**.

☐ Let me select individual events.

☒ **Active**

We will deliver event details when this hook is triggered.

Add webhook

© 2024 GitHub, Inc. Terms Privacy Security Status Docs Contact Manage cookies Do not share my personal information

← → ↻ Kishorgujar / DevOps-CICD  + 🔍 📄 📧

<> Code Issues Pull requests Actions Projects Wiki Security Insights **Settings**

General

Access

- Collaborators
- Moderation options

Code and automation

- Branches
- Tags
- Rules
- Actions
- Webhooks**
- Environments
- Codespaces
- Pages

Security

- Code security

### Webhooks

Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#).

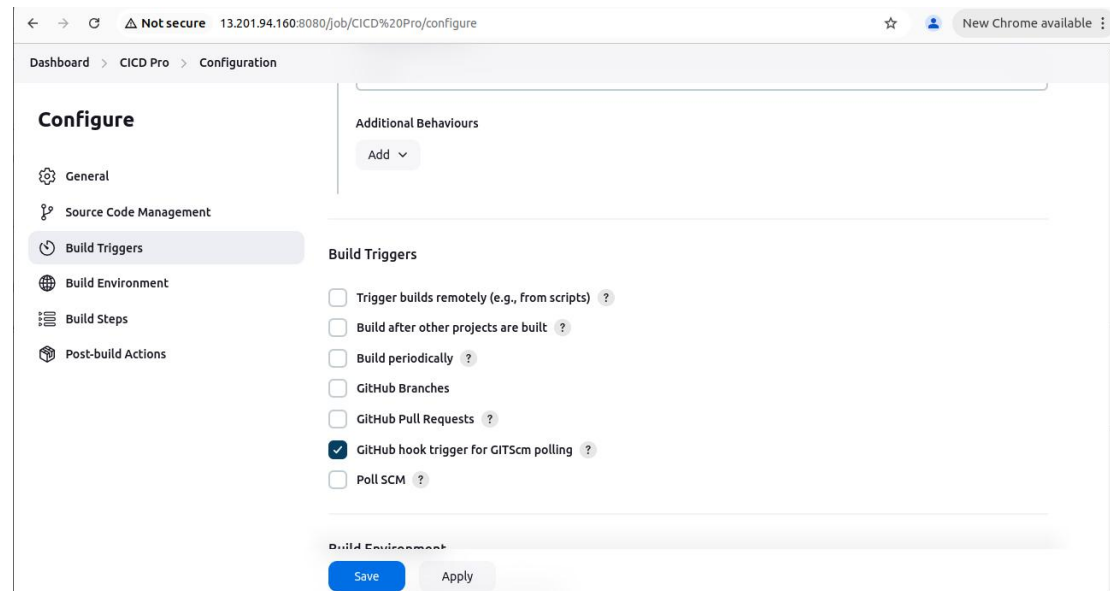
[Add webhook](#)

✓ <http://13.201.94.160:8080/github-w...> (all events)

Edit Delete

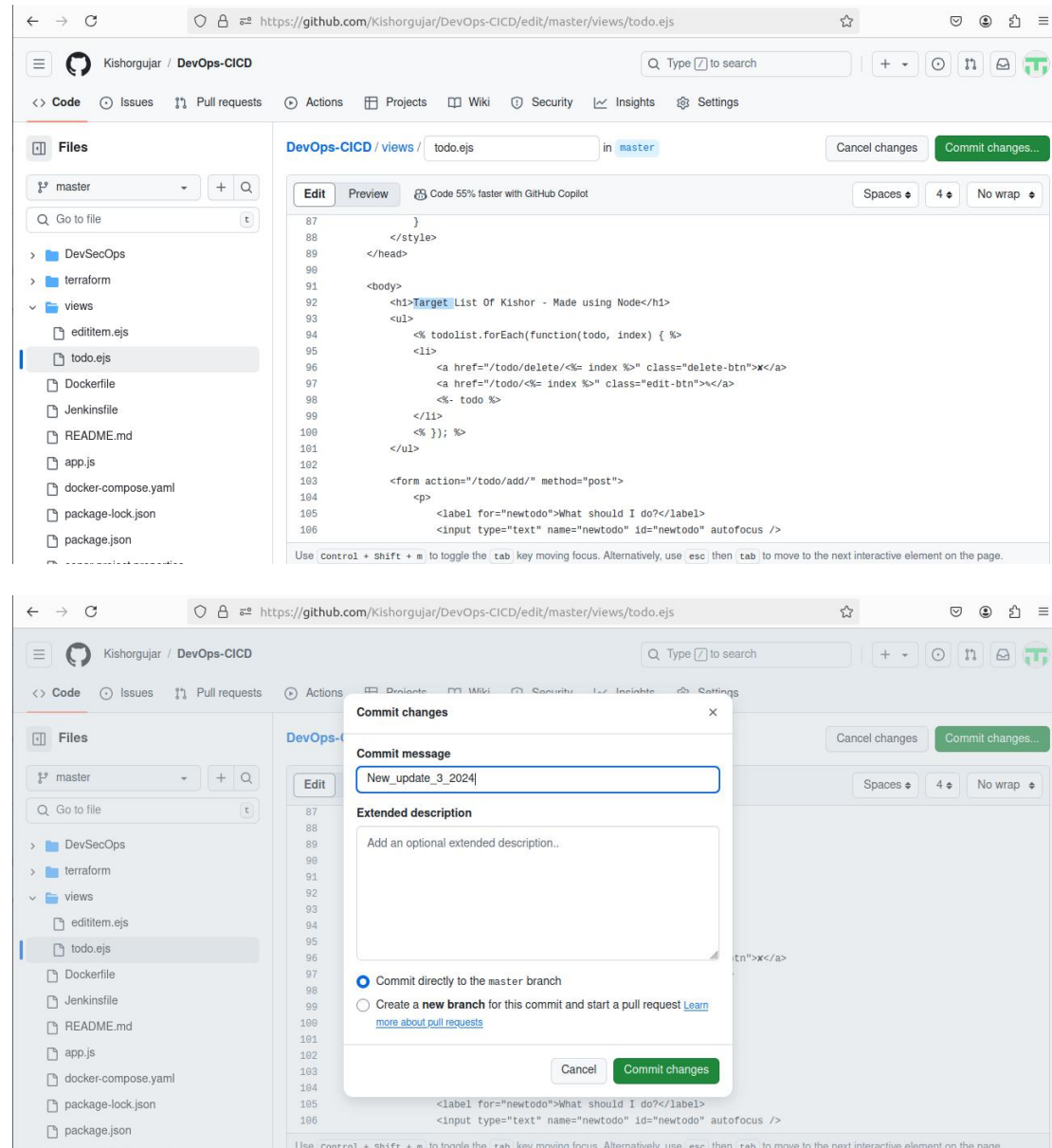
Last delivery was successful.

In Jenkins, select **GitHub hook trigger for GITScm polling** to enable automatic build triggering whenever changes are pushed to your GitHub repository. This allows Jenkins to listen for webhook notifications from GitHub, ensuring your builds are always up-to-date with the latest code.

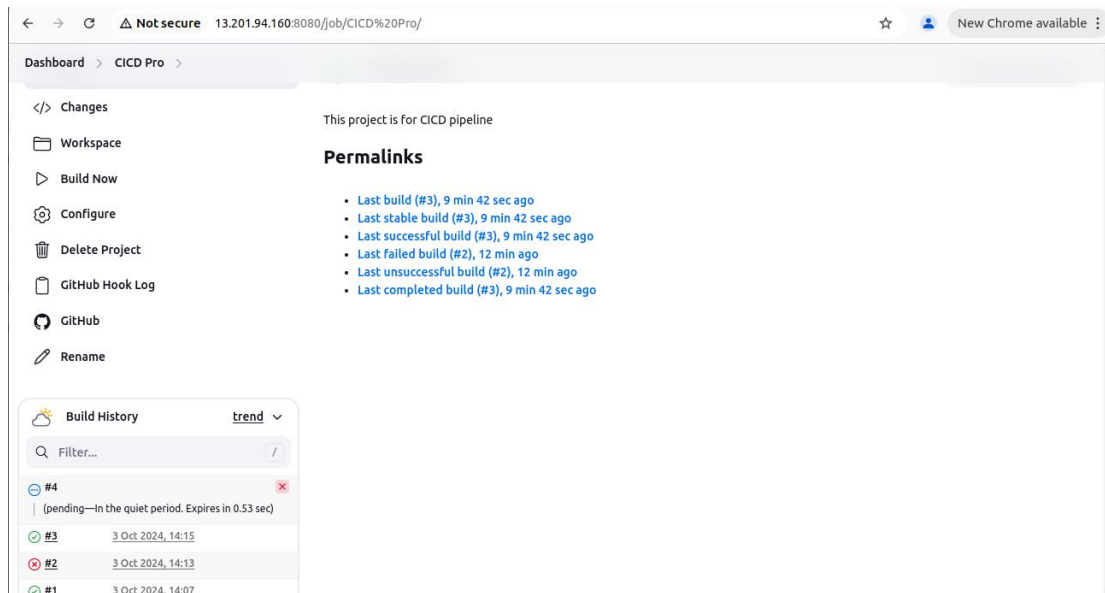


### 3] Change Code & Verify Output

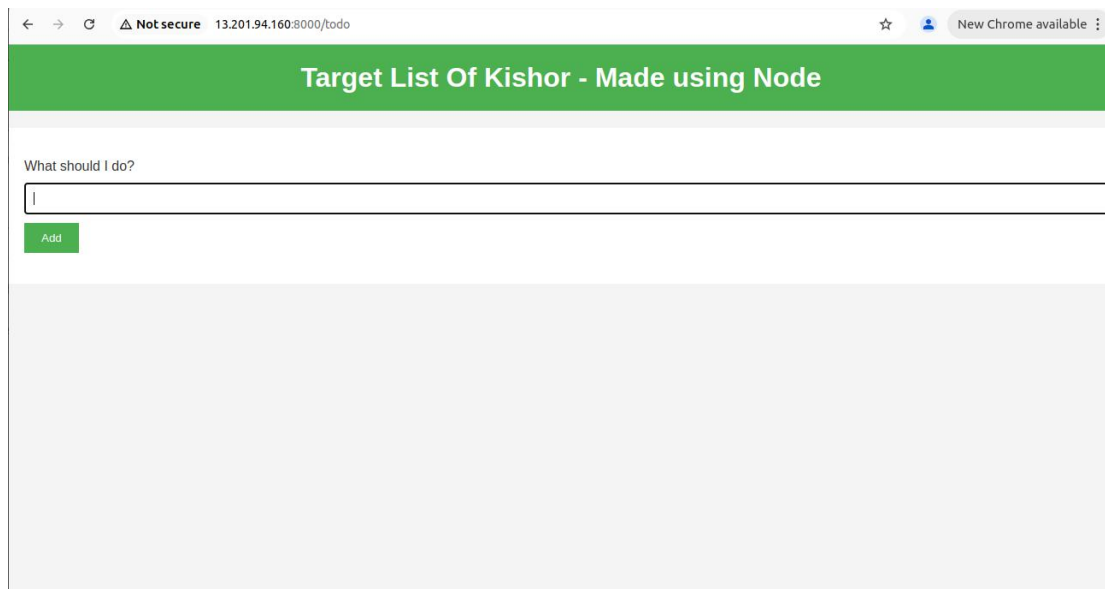
To verify your CI/CD process, go to your GitHub repository and open the `todo.ejs` file. Make a small change, such as modifying the header from "TODO" to "Target." After making the change, commit it to the repository. This will trigger Jenkins to start the build process automatically, demonstrating the integration between GitHub and Jenkins.



Now, return to the Jenkins web page, where you'll see that the build process has started automatically without any manual intervention. Look for build number "#4", which indicates that Jenkins has triggered the build in response to your recent commit. This seamless automation showcases the power of CI/CD, allowing for efficient and rapid deployment of code changes.



Next, open your web browser and enter your instance's **public IP** followed by `:8000`. You'll notice that the application header has changed, reflecting the modification you made in the GitHub code. This entire process happened without any manual intervention—just by updating the code in GitHub. Jenkins automatically triggered the build, testing, and deployment steps, demonstrating the effectiveness of the CI/CD pipeline in action.



## Conclusion

This CI/CD pipeline for the Node.js Todo List application not only streamlines the development process but also demonstrates my capability to implement modern DevOps practices. By leveraging tools like Jenkins and Docker, I have created an efficient workflow that enhances code quality and deployment speed, contributing to a more agile development environment.