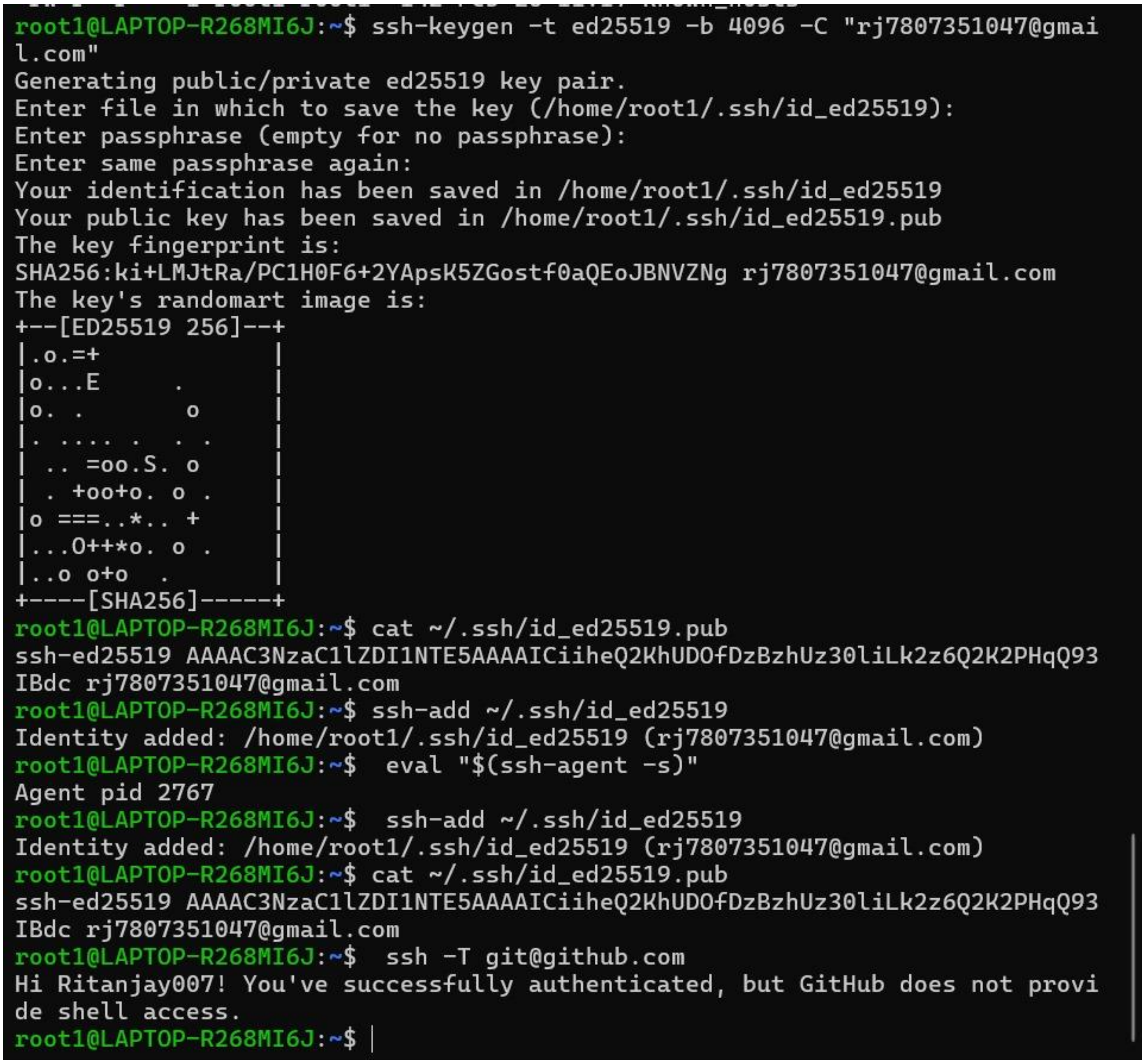
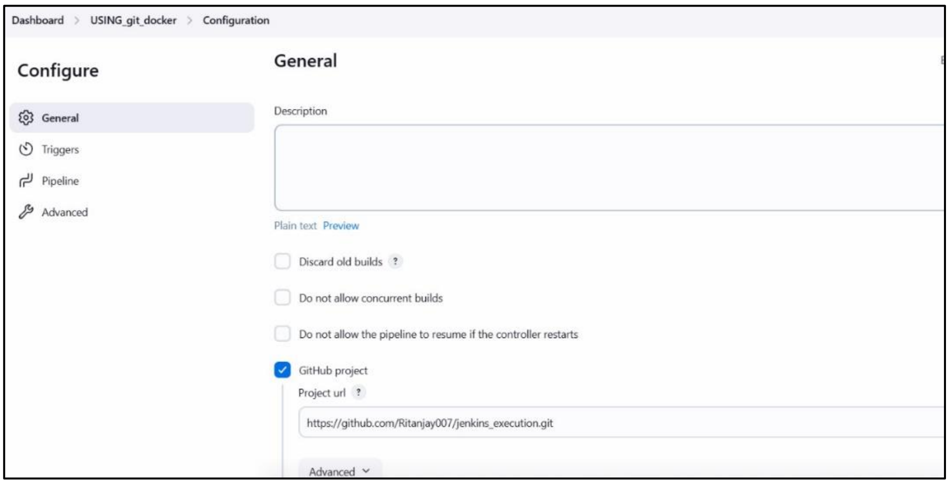
To create an ssh key and link it to github:



```
root1@LAPTOP-R268MI6J:~$ ssh-keygen -t ed25519 -b 4096 -C "rj7807351047@gmai
l.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/root1/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/root1/.ssh/id_ed25519
Your public key has been saved in /home/root1/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:ki+LMJtRa/PC1H0F6+2YApsK5ZGostf0aQEoJBNVZNg rj7807351047@gmail.com
The key's randomart image is:
+--[ED25519 256]--+
|.o.=+            |
|o...E      .     |
|o. .        o    |
|. .... .   . .   |
| .. =oo.S. o     |
|  . +oo+o. o .   |
|o ===..*.. +     |
|...O++*o. o .    |
|..o o+o  .       |
+----[SHA256]-----+
root1@LAPTOP-R268MI6J:~$ cat ~/.ssh/id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAICiiheQ2KhUDOfDzBzhUz30liLk2z6Q2K2PHqQ93
IBdc rj7807351047@gmail.com
root1@LAPTOP-R268MI6J:~$ ssh-add ~/.ssh/id_ed25519
Identity added: /home/root1/.ssh/id_ed25519 (rj7807351047@gmail.com)
root1@LAPTOP-R268MI6J:~$  eval "$(ssh-agent -s)"
Agent pid 2767
root1@LAPTOP-R268MI6J:~$  ssh-add ~/.ssh/id_ed25519
Identity added: /home/root1/.ssh/id_ed25519 (rj7807351047@gmail.com)
root1@LAPTOP-R268MI6J:~$ cat ~/.ssh/id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAICiiheQ2KhUDOfDzBzhUz30liLk2z6Q2K2PHqQ93
IBdc rj7807351047@gmail.com
root1@LAPTOP-R268MI6J:~$  ssh -T git@github.com
Hi Ritanjay007! You've successfully authenticated, but GitHub does not provi
de shell access.
root1@LAPTOP-R268MI6J:~$
```

After adding the key in system then we cat it and copy and paste it in github ssh and keys.



General

Description

Plain text Preview

□ Discard old builds

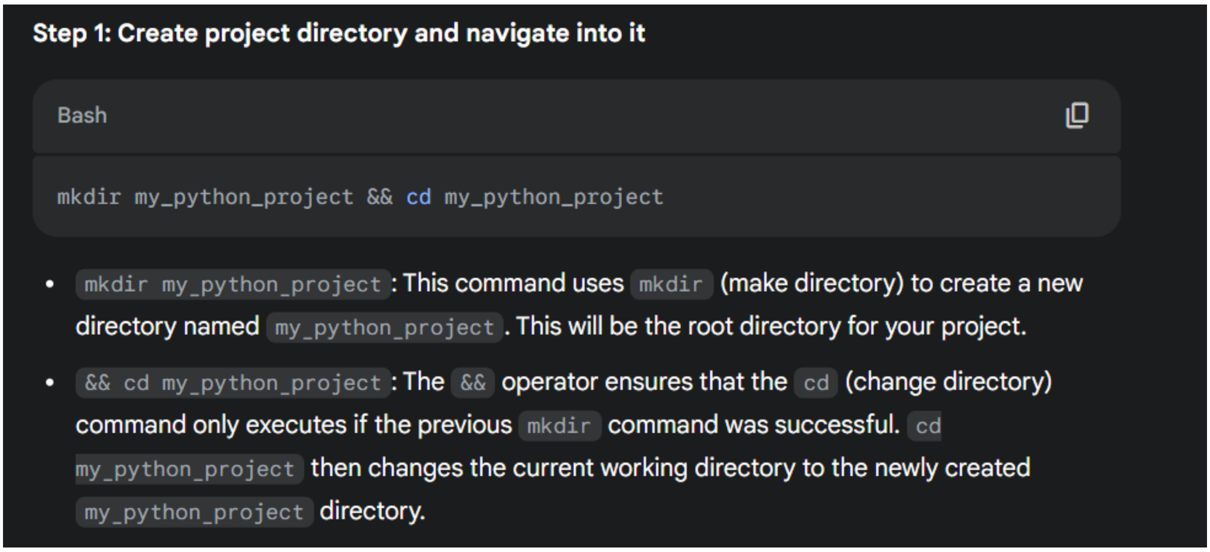
□ Do not allow concurrent builds

□ Do not allow the pipeline to resume if the controller restarts

☑ GitHub project

Project url

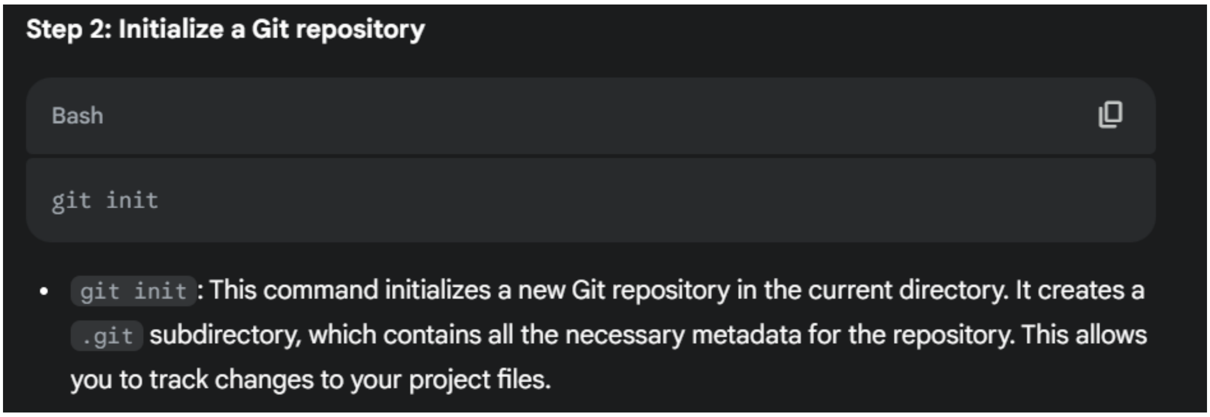
https://github.com/Ritanjay007/jenkins_execution.git



Step 1: Create project directory and navigate into it

```bash
mkdir my_python_project && cd my_python_project
```

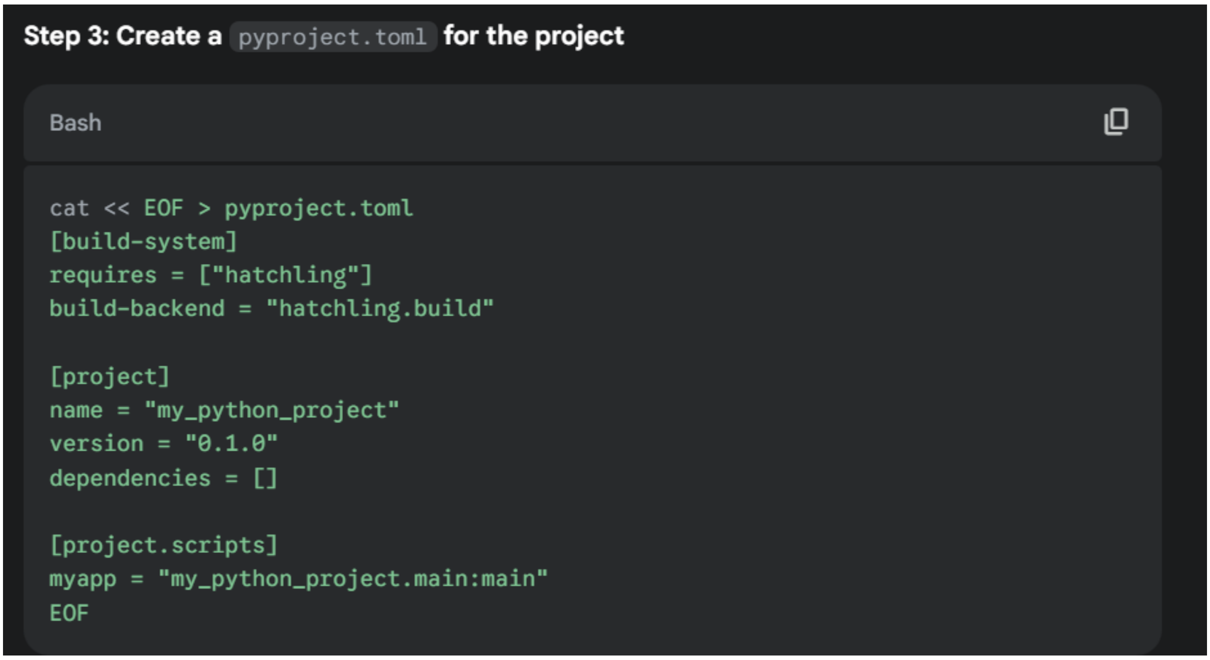
- `mkdir my_python_project`: This command uses `mkdir` (make directory) to create a new directory named `my_python_project`. This will be the root directory for your project.
- `&& cd my_python_project`: The `&&` operator ensures that the `cd` (change directory) command only executes if the previous `mkdir` command was successful. `cd my_python_project` then changes the current working directory to the newly created `my_python_project` directory.



Step 2: Initialize a Git repository

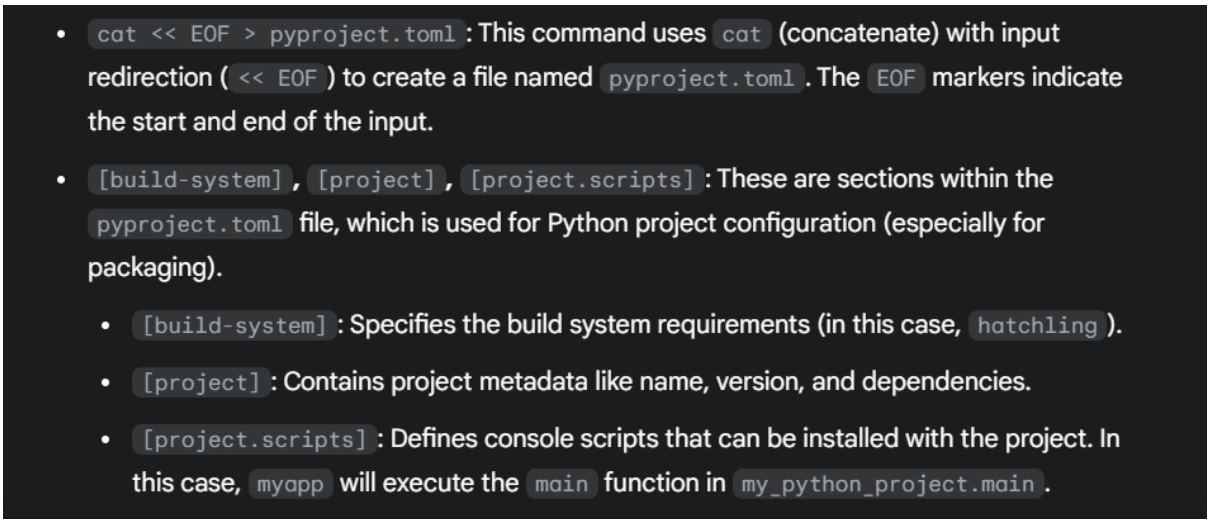
```bash
git init
```

- `git init`: This command initializes a new Git repository in the current directory. It creates a `.git` subdirectory, which contains all the necessary metadata for the repository. This allows you to track changes to your project files.

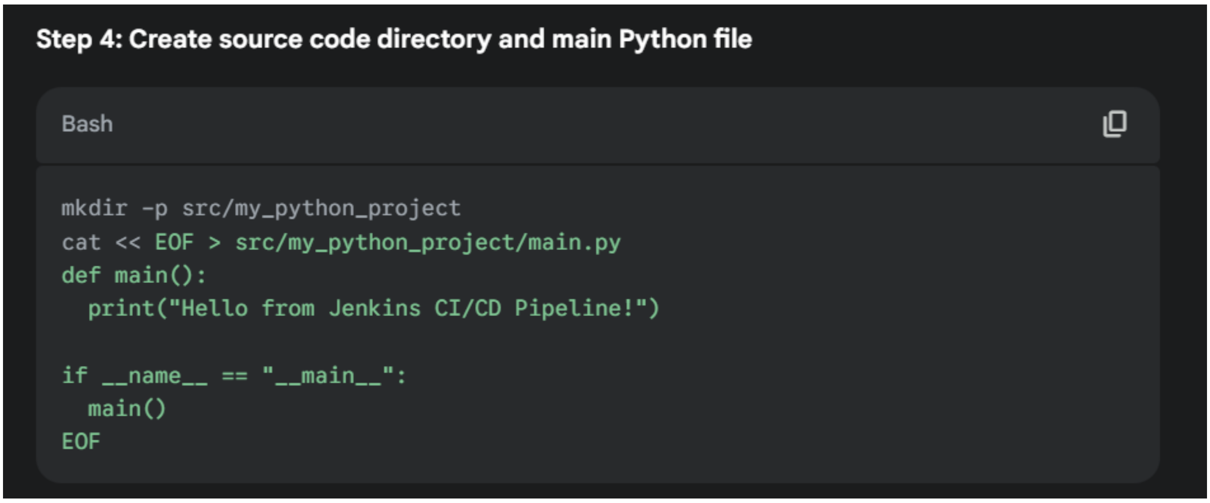


Step 3: Create a `pyproject.toml` for the project

```bash
cat << EOF > pyproject.toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "my_python_project"
version = "0.1.0"
dependencies = []

[project.scripts]
myapp = "my_python_project.main:main"
EOF
```



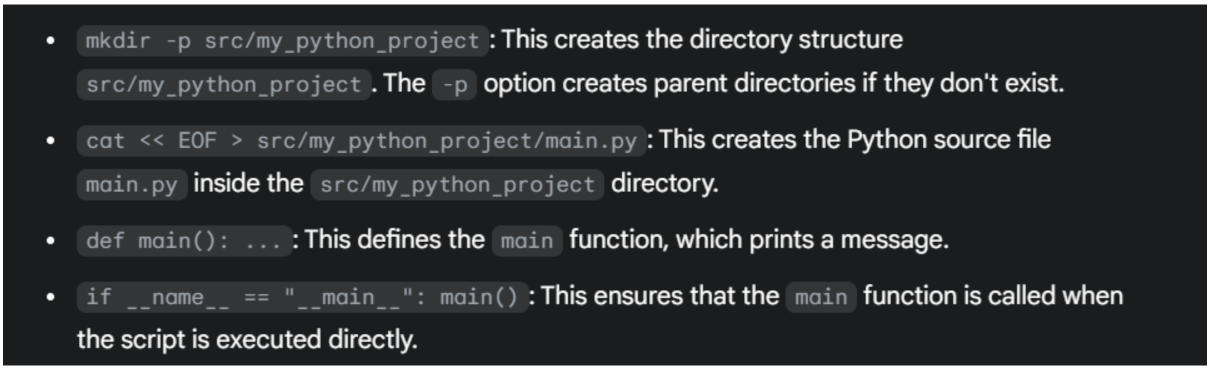
- `cat << EOF > pyproject.toml`: This command uses `cat` (concatenate) with input redirection (`<< EOF`) to create a file named `pyproject.toml`. The `EOF` markers indicate the start and end of the input.
- `[build-system]`, `[project]`, `[project.scripts]`: These are sections within the `pyproject.toml` file, which is used for Python project configuration (especially for packaging).
  - `[build-system]`: Specifies the build system requirements (in this case, `hatchling`).
  - `[project]`: Contains project metadata like name, version, and dependencies.
  - `[project.scripts]`: Defines console scripts that can be installed with the project. In this case, `myapp` will execute the `main` function in `my_python_project.main`.



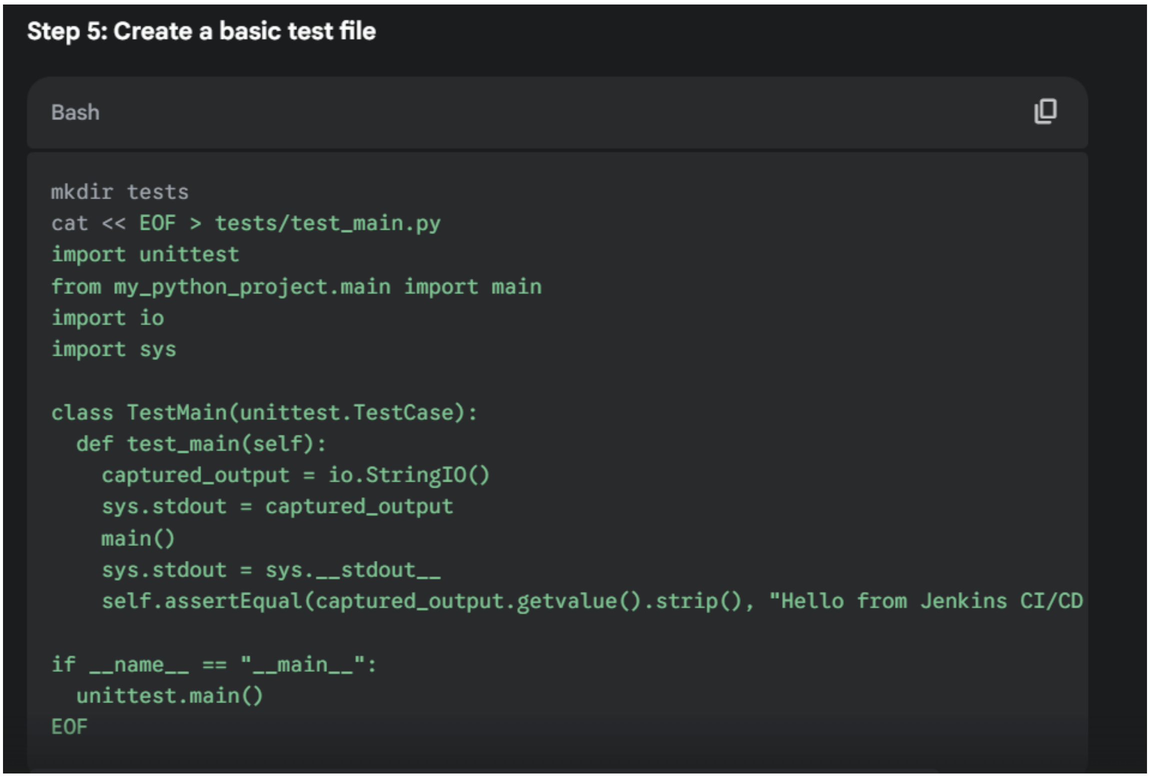
Step 4: Create source code directory and main Python file

Bash

```bash
mkdir -p src/my_python_project
cat << EOF > src/my_python_project/main.py
def main():
  print("Hello from Jenkins CI/CD Pipeline!")

if __name__ == "__main__":
  main()
EOF
```

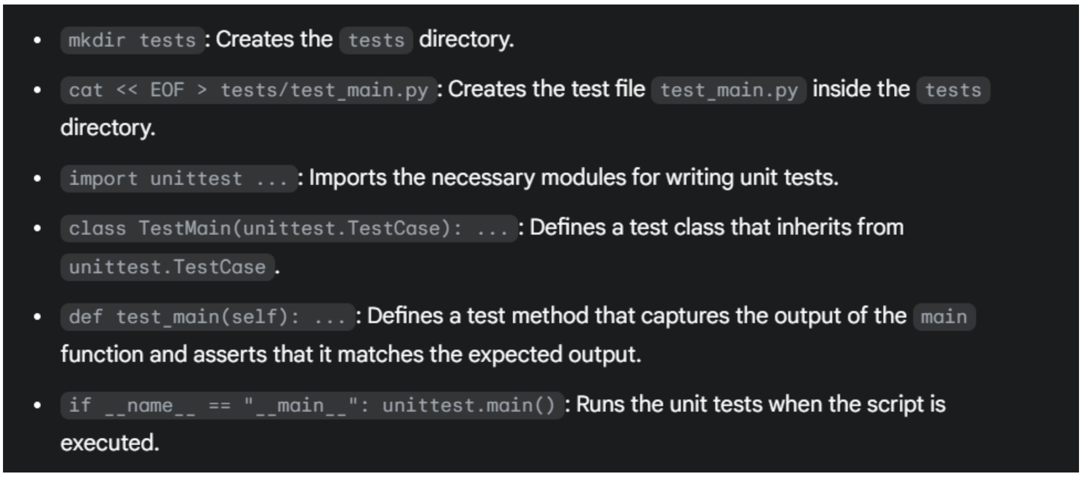


- `mkdir -p src/my_python_project`: This creates the directory structure `src/my_python_project`. The `-p` option creates parent directories if they don't exist.
- `cat << EOF > src/my_python_project/main.py`: This creates the Python source file `main.py` inside the `src/my_python_project` directory.
- `def main(): ...`: This defines the `main` function, which prints a message.
- `if __name__ == "__main__": main()`: This ensures that the `main` function is called when the script is executed directly.

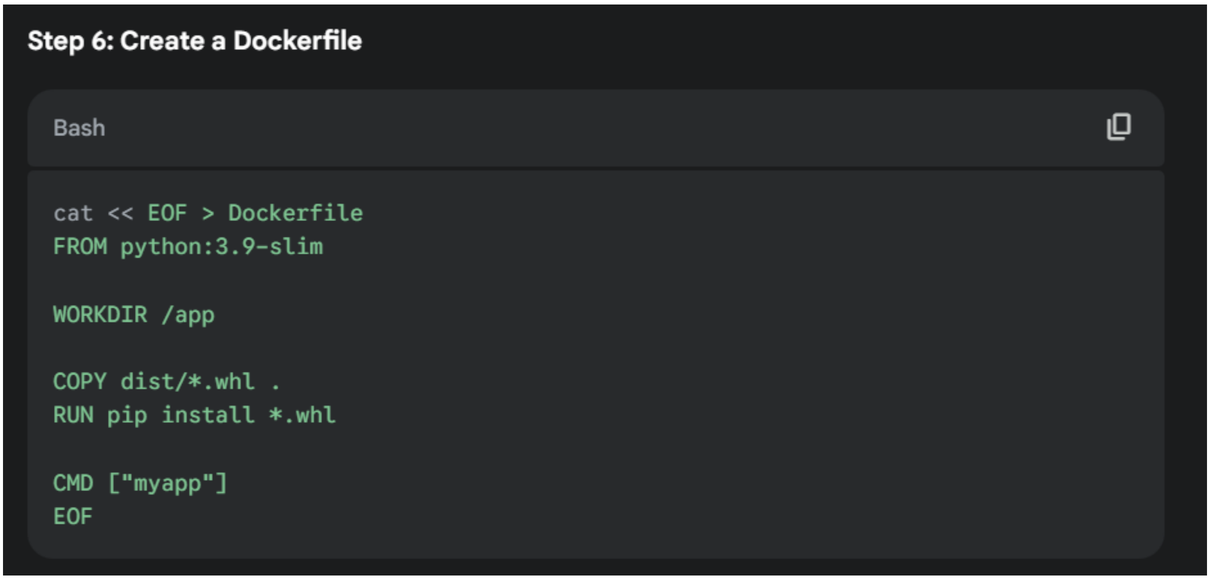


Step 5: Create a basic test file

```bash
mkdir tests
cat << EOF > tests/test_main.py
import unittest
from my_python_project.main import main
import io
import sys

class TestMain(unittest.TestCase):
  def test_main(self):
    captured_output = io.StringIO()
    sys.stdout = captured_output
    main()
    sys.stdout = sys.__stdout__
    self.assertEqual(captured_output.getvalue().strip(), "Hello from Jenkins CI/CD

if __name__ == "__main__":
  unittest.main()
EOF
```

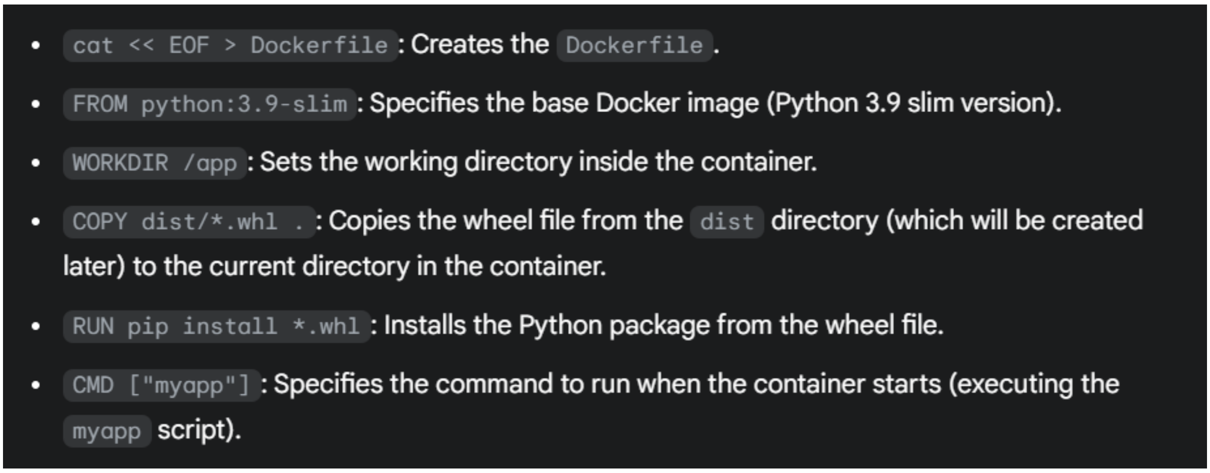


- `mkdir tests`: Creates the `tests` directory.
- `cat << EOF > tests/test_main.py`: Creates the test file `test_main.py` inside the `tests` directory.
- `import unittest ...`: Imports the necessary modules for writing unit tests.
- `class TestMain(unittest.TestCase): ...`: Defines a test class that inherits from `unittest.TestCase`.
- `def test_main(self): ...`: Defines a test method that captures the output of the `main` function and asserts that it matches the expected output.
- `if __name__ == "__main__": unittest.main()`: Runs the unit tests when the script is executed.

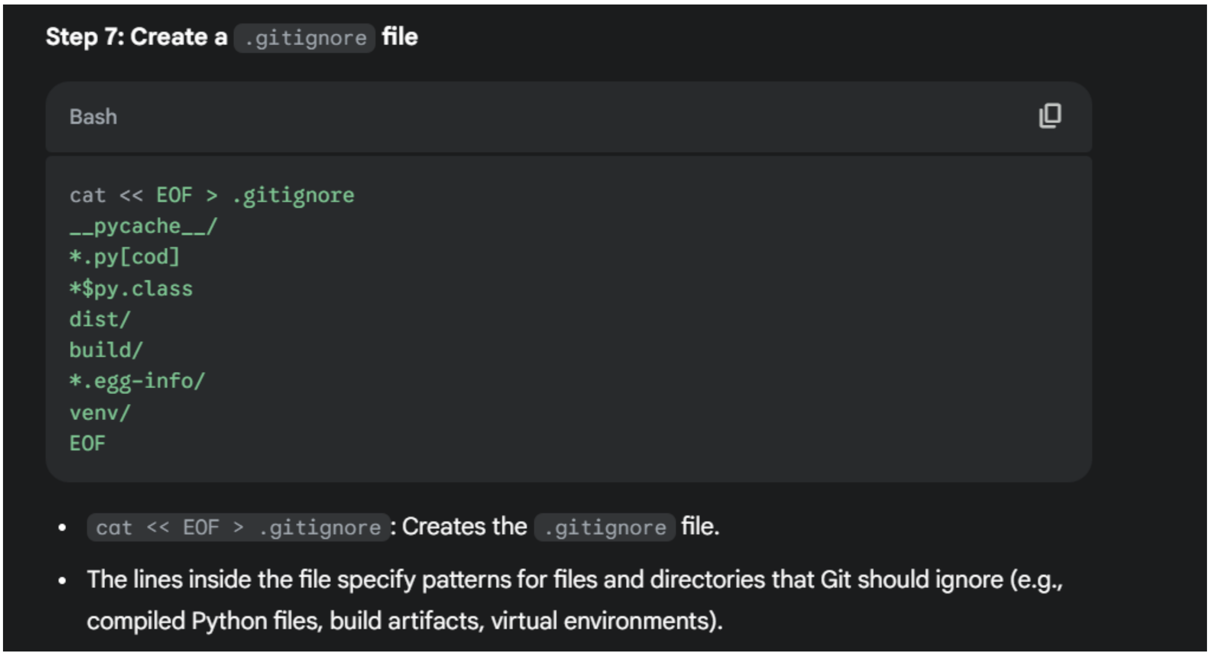


Step 6: Create a Dockerfile

```bash
cat << EOF > Dockerfile
FROM python:3.9-slim

WORKDIR /app

COPY dist/*.whl .
RUN pip install *.whl

CMD ["myapp"]
EOF
```



- `cat << EOF > Dockerfile`: Creates the `Dockerfile`.
- `FROM python:3.9-slim`: Specifies the base Docker image (Python 3.9 slim version).
- `WORKDIR /app`: Sets the working directory inside the container.
- `COPY dist/*.whl .`: Copies the wheel file from the `dist` directory (which will be created later) to the current directory in the container.
- `RUN pip install *.whl`: Installs the Python package from the wheel file.
- `CMD ["myapp"]`: Specifies the command to run when the container starts (executing the `myapp` script).

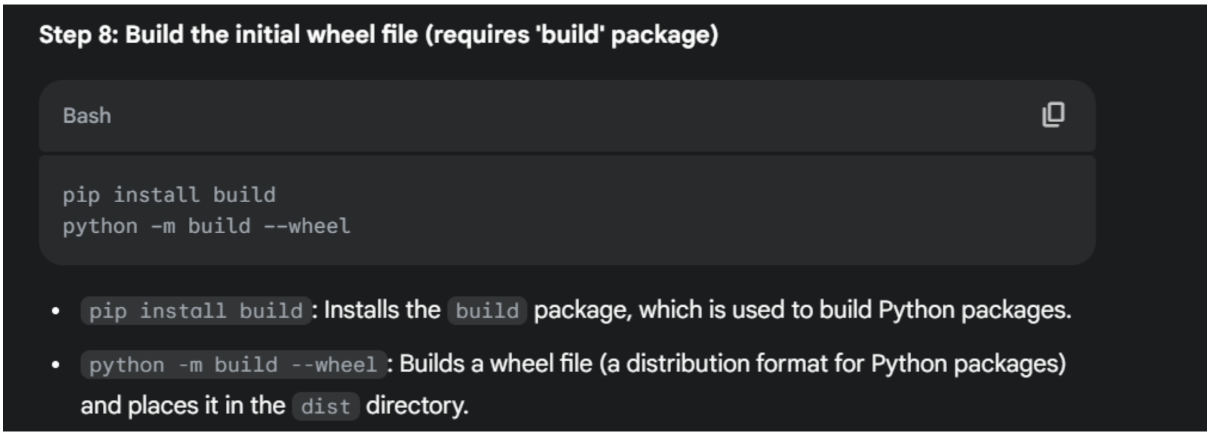


Step 7: Create a `.gitignore` file

Bash

```bash
cat << EOF > .gitignore
__pycache__/
*.py[cod]
*$py.class
dist/
build/
*.egg-info/
venv/
EOF
```

- `cat << EOF > .gitignore`: Creates the `.gitignore` file.
- The lines inside the file specify patterns for files and directories that Git should ignore (e.g., compiled Python files, build artifacts, virtual environments).

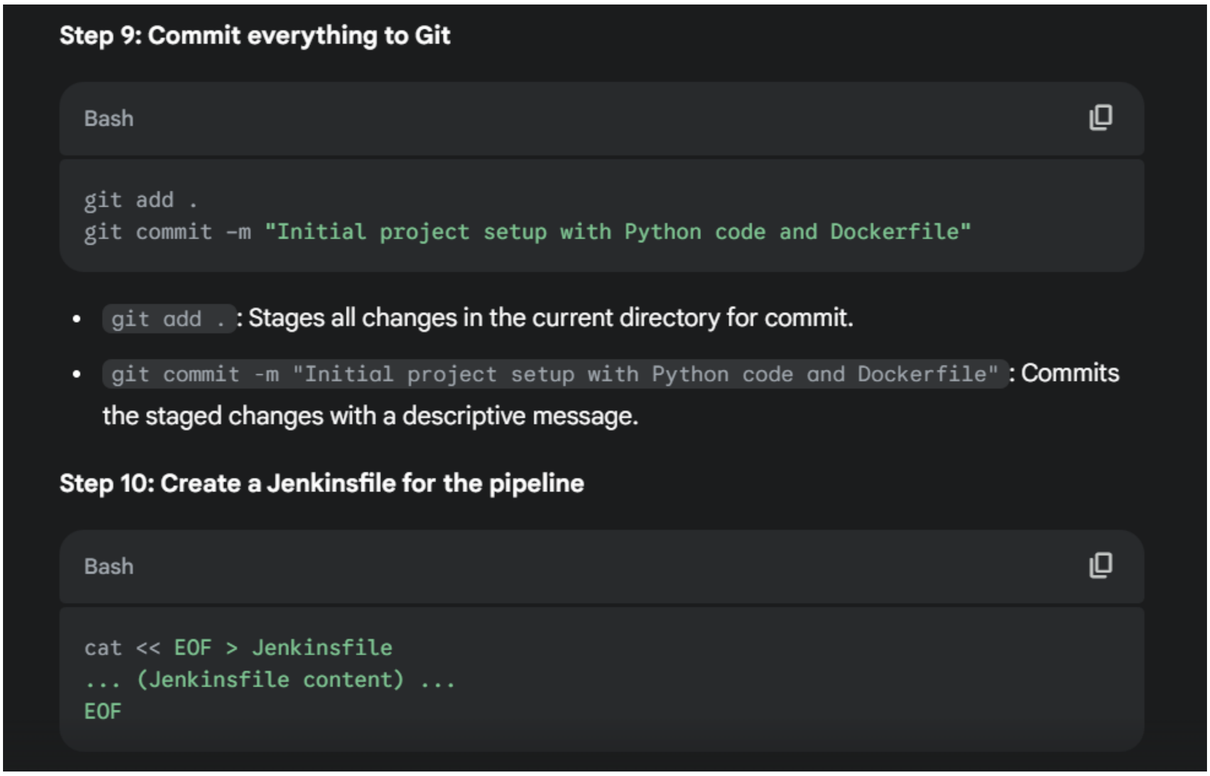


Step 8: Build the initial wheel file (requires 'build' package)

Bash

```bash
pip install build
python -m build --wheel
```

- `pip install build`: Installs the `build` package, which is used to build Python packages.
- `python -m build --wheel`: Builds a wheel file (a distribution format for Python packages) and places it in the `dist` directory.



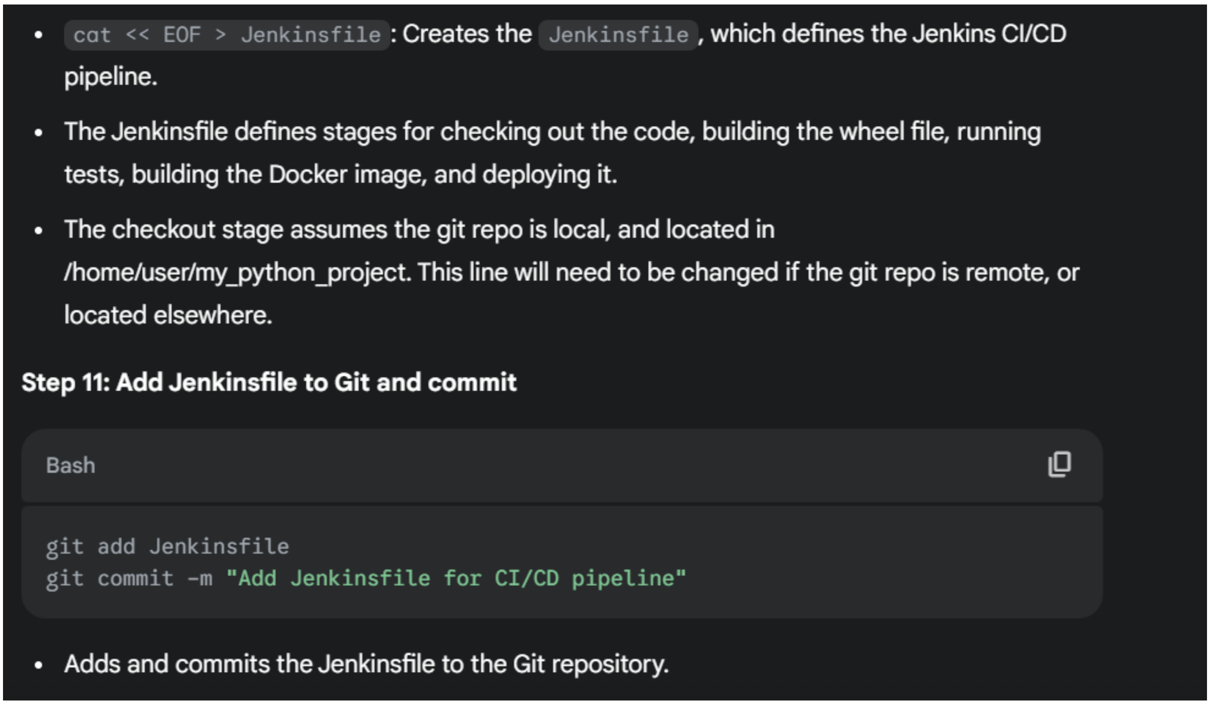
Step 9: Commit everything to Git

```bash
git add .
git commit -m "Initial project setup with Python code and Dockerfile"
```

- `git add .`: Stages all changes in the current directory for commit.
- `git commit -m "Initial project setup with Python code and Dockerfile"`: Commits the staged changes with a descriptive message.

Step 10: Create a Jenkinsfile for the pipeline

```bash
cat << EOF > Jenkinsfile
... (Jenkinsfile content) ...
EOF
```



- `cat << EOF > Jenkinsfile`: Creates the `Jenkinsfile`, which defines the Jenkins CI/CD pipeline.
- The Jenkinsfile defines stages for checking out the code, building the wheel file, running tests, building the Docker image, and deploying it.
- The checkout stage assumes the git repo is local, and located in /home/user/my_python_project. This line will need to be changed if the git repo is remote, or located elsewhere.

Step 11: Add Jenkinsfile to Git and commit

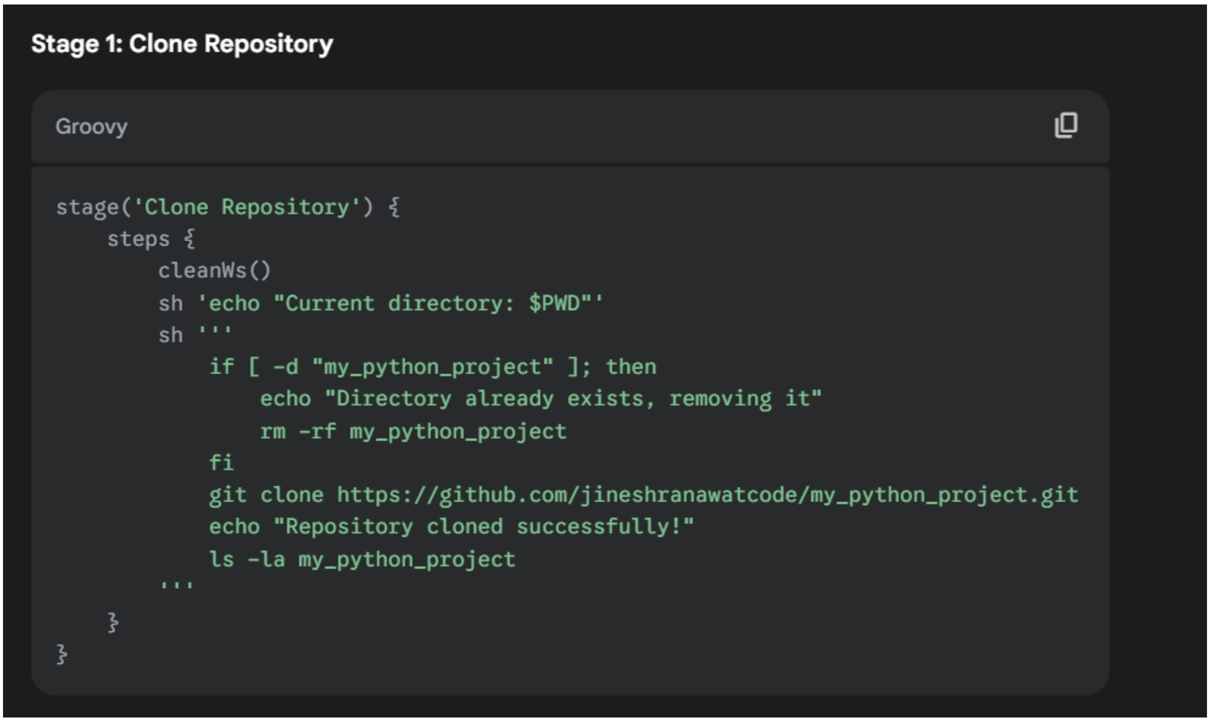
```bash
git add Jenkinsfile
git commit -m "Add Jenkinsfile for CI/CD pipeline"
```

- Adds and commits the Jenkinsfile to the Git repository.

After this we use command:

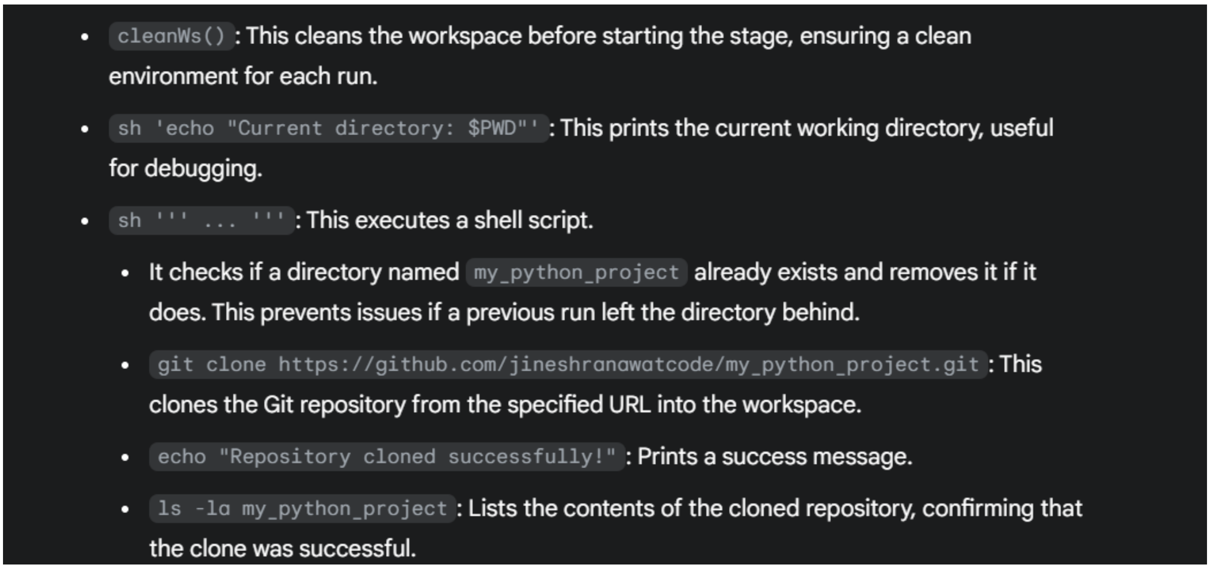
Sudo systemctl start Jenkins

To start Jenkins and after that we go to browser and open the local host to open Jenkins. And within Jenkins file we create a new file and select coding type as pipeline and add project type as github and add our github code for the repository.

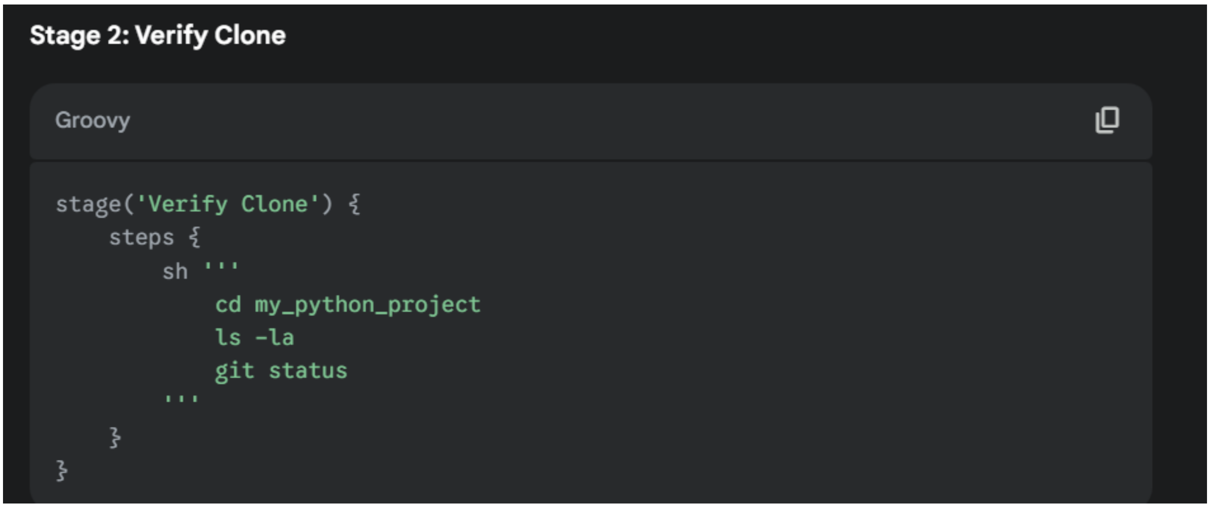


Stage 1: Clone Repository

```
stage('Clone Repository') {
    steps {
        cleanWs()
        sh 'echo "Current directory: $PWD"'
        sh '''
            if [ -d "my_python_project" ]; then
                echo "Directory already exists, removing it"
                rm -rf my_python_project
            fi
            git clone https://github.com/jineshranawatcode/my_python_project.git
            echo "Repository cloned successfully!"
            ls -la my_python_project
        '''
    }
}
```

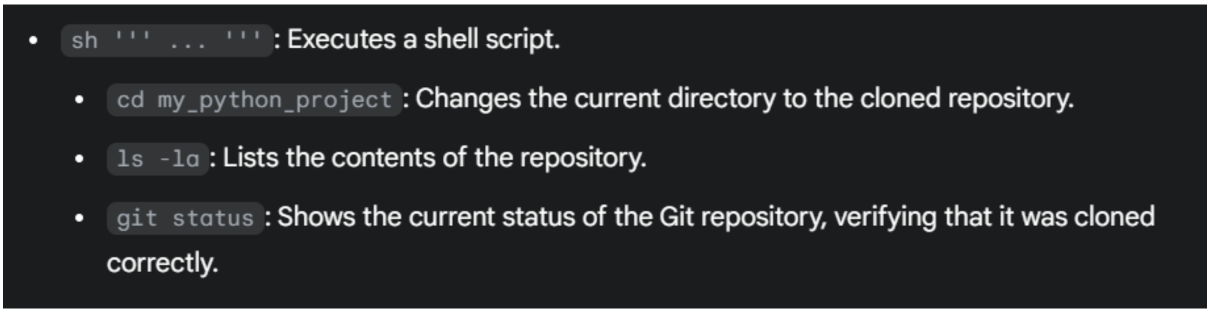


- `cleanWs()`: This cleans the workspace before starting the stage, ensuring a clean environment for each run.
- `sh 'echo "Current directory: $PWD"'`: This prints the current working directory, useful for debugging.
- `sh ''' ... '''`: This executes a shell script.
    - It checks if a directory named `my_python_project` already exists and removes it if it does. This prevents issues if a previous run left the directory behind.
    - `git clone https://github.com/jineshranawatcode/my_python_project.git`: This clones the Git repository from the specified URL into the workspace.
    - `echo "Repository cloned successfully!"`: Prints a success message.
    - `ls -la my_python_project`: Lists the contents of the cloned repository, confirming that the clone was successful.

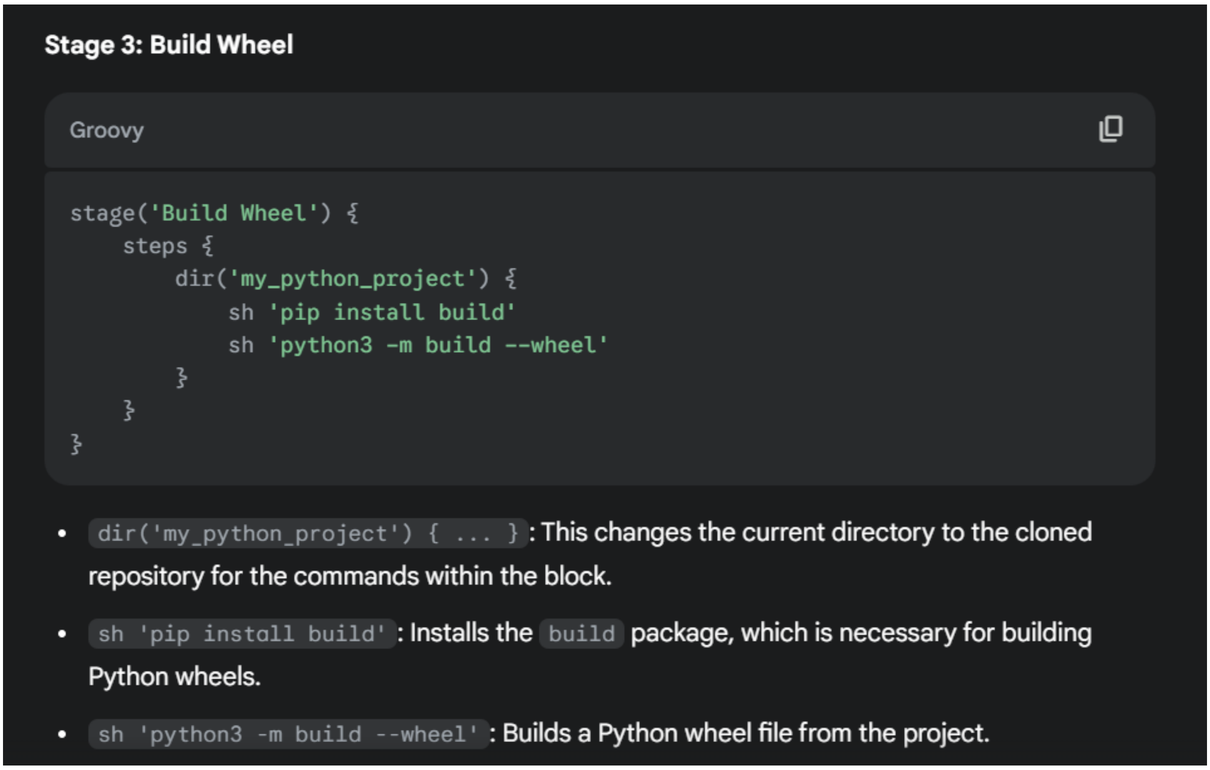


Stage 2: Verify Clone

```groovy
stage('Verify Clone') {
    steps {
        sh '''
            cd my_python_project
            ls -la
            git status
        '''
    }
}
```



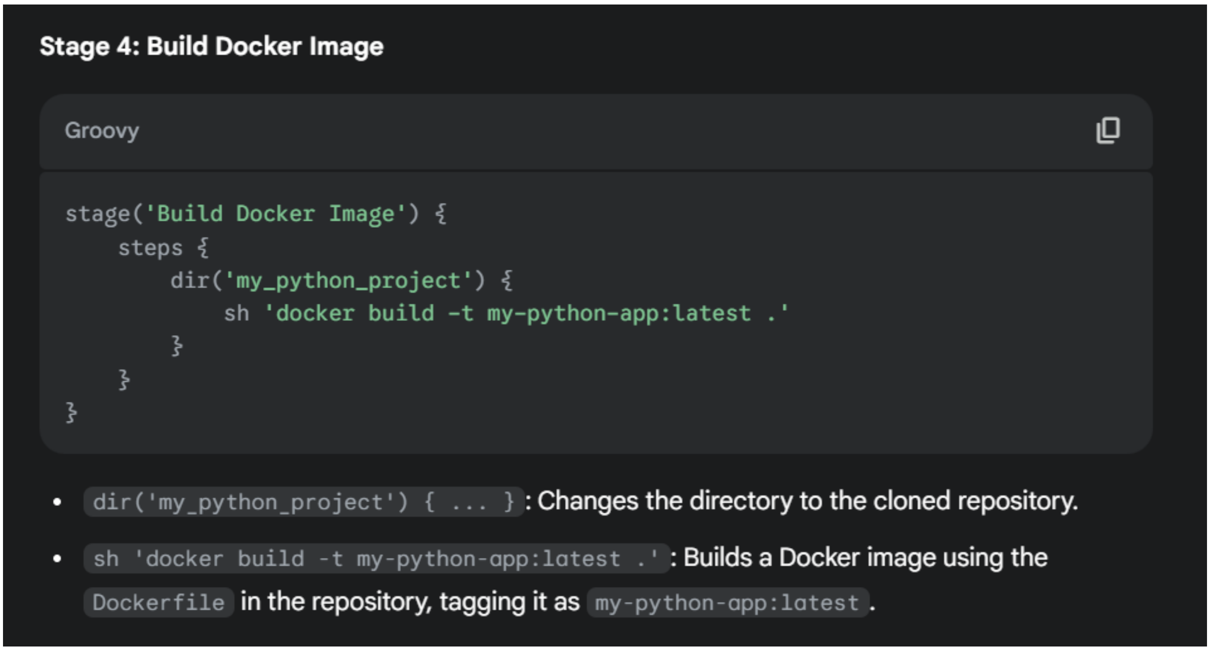
- `sh ''' ... '''`: Executes a shell script.
  - `cd my_python_project`: Changes the current directory to the cloned repository.
  - `ls -la`: Lists the contents of the repository.
  - `git status`: Shows the current status of the Git repository, verifying that it was cloned correctly.



Stage 3: Build Wheel

```groovy
stage('Build Wheel') {
    steps {
        dir('my_python_project') {
            sh 'pip install build'
            sh 'python3 -m build --wheel'
        }
    }
}
```

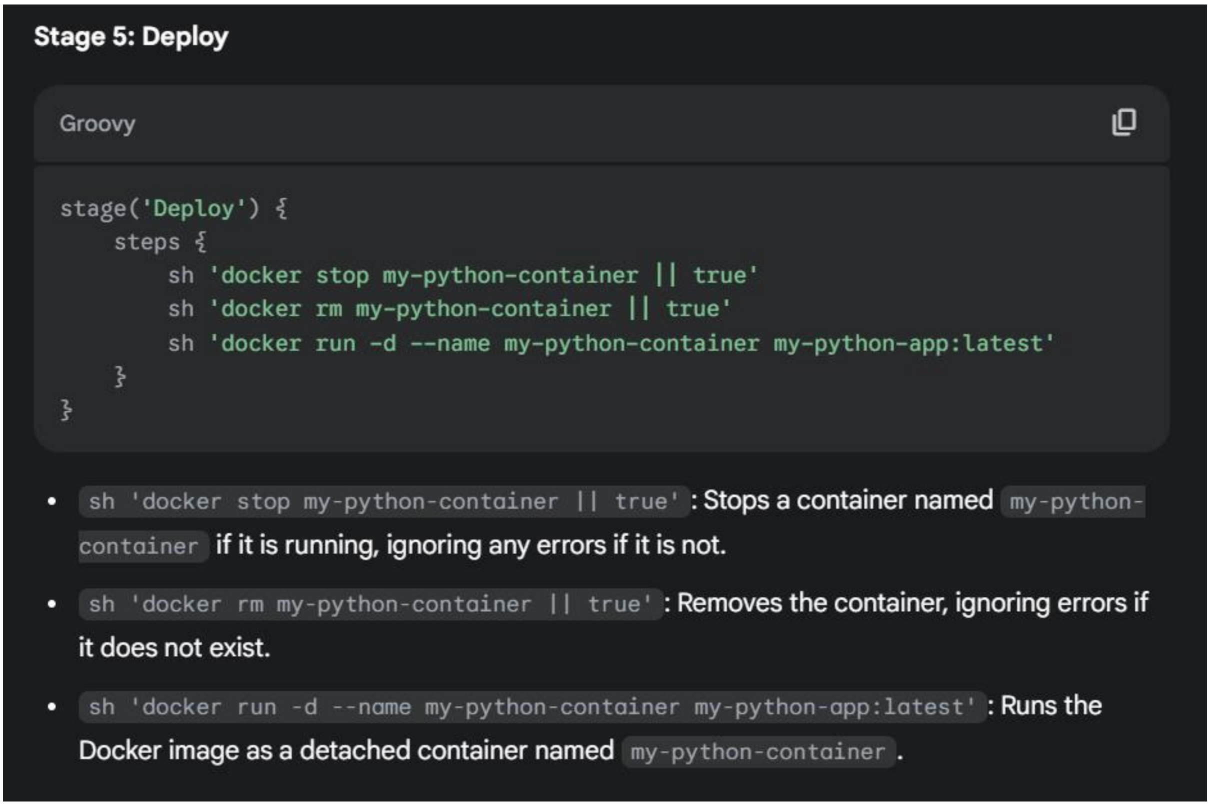
- `dir('my_python_project') { ... }`: This changes the current directory to the cloned repository for the commands within the block.
- `sh 'pip install build'`: Installs the `build` package, which is necessary for building Python wheels.
- `sh 'python3 -m build --wheel'`: Builds a Python wheel file from the project.



Stage 4: Build Docker Image

```groovy
stage('Build Docker Image') {
    steps {
        dir('my_python_project') {
            sh 'docker build -t my-python-app:latest .'
        }
    }
}
```

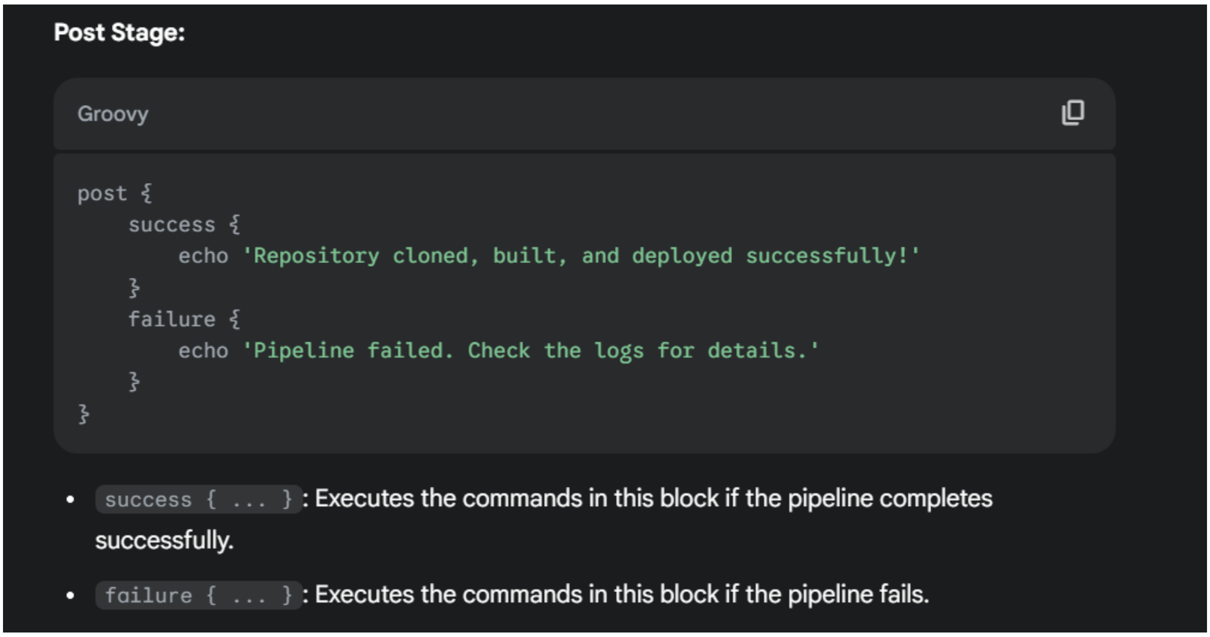
- `dir('my_python_project') { ... }`: Changes the directory to the cloned repository.
- `sh 'docker build -t my-python-app:latest .'`: Builds a Docker image using the `Dockerfile` in the repository, tagging it as `my-python-app:latest`.



Stage 5: Deploy

```groovy
stage('Deploy') {
    steps {
        sh 'docker stop my-python-container || true'
        sh 'docker rm my-python-container || true'
        sh 'docker run -d --name my-python-container my-python-app:latest'
    }
}
```

- `sh 'docker stop my-python-container || true'`: Stops a container named `my-python-container` if it is running, ignoring any errors if it is not.
- `sh 'docker rm my-python-container || true'`: Removes the container, ignoring errors if it does not exist.
- `sh 'docker run -d --name my-python-container my-python-app:latest'`: Runs the Docker image as a detached container named `my-python-container`.

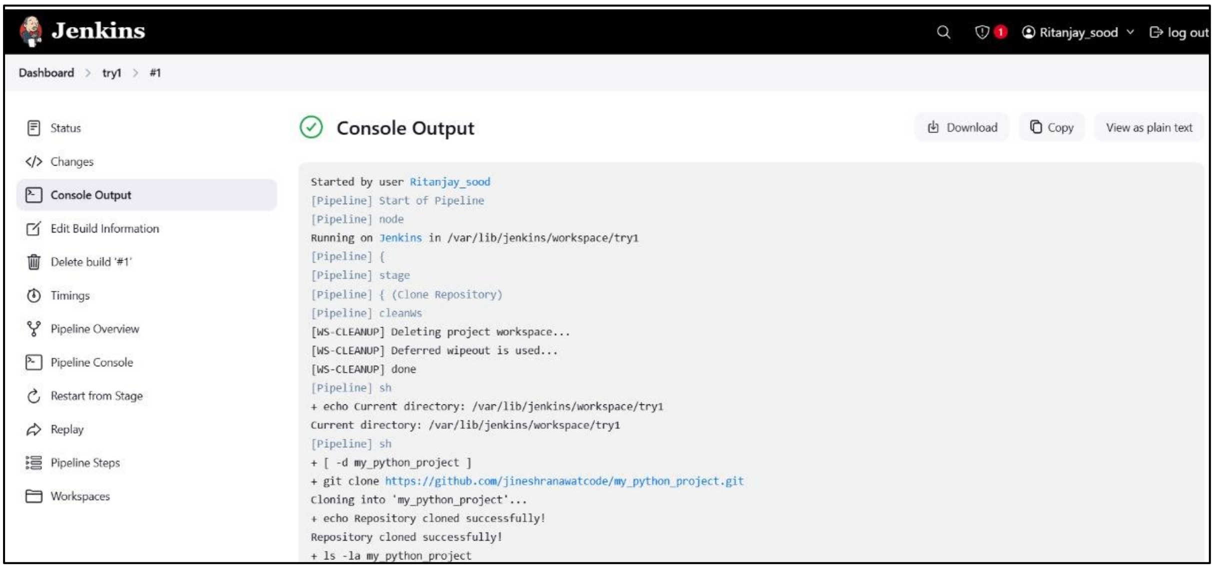


Post Stage:

Groovy

```groovy
post {
    success {
        echo 'Repository cloned, built, and deployed successfully!'
    }
    failure {
        echo 'Pipeline failed. Check the logs for details.'
    }
}
```

- `success { ... }`: Executes the commands in this block if the pipeline completes successfully.
- `failure { ... }`: Executes the commands in this block if the pipeline fails.



Console Output

```
Started by user Ritanjay_sood
[Pipeline] Start of Pipeline
[Pipeline] node
Running on Jenkins in /var/lib/jenkins/workspace/try1
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Clone Repository)
[Pipeline] cleanWs
[WS-CLEANUP] Deleting project workspace...
[WS-CLEANUP] Deferred wipeout is used...
[WS-CLEANUP] done
[Pipeline] sh
+ echo Current directory: /var/lib/jenkins/workspace/try1
Current directory: /var/lib/jenkins/workspace/try1
[Pipeline] sh
+ [ -d my_python_project ]
+ git clone https://github.com/jineshranawatcode/my_python_project.git
Cloning into 'my_python_project'...
+ echo Repository cloned successfully!
Repository cloned successfully!
+ ls -la my_python_project
```