Experiment Steps :

Step 1: Set Up the Web Application and Local HTTP Server

- Create a simple web application or use an existing one. Ensure it can be hosted by a local HTTP server.
- Set up a local HTTP server (e.g., Apache or Nginx) to serve the web application. Ensure it's configured correctly and running.

Step 2: Set Up a Git Repository

Create a Git repository for your web application. Initialize it with the following commands:

git init git add .

git commit -m "Initial commit"

Create a remote Git repository (e.g., on GitHub or Bitbucket) to push your code to later.

Step 3: Install and Configure Jenkins

- Download and install Jenkins following the instructions for your operating system (https://www.jenkins.io/download/).
- Open Jenkins in your web browser (usually at http://localhost:8080) and complete the initial setup.
- Install the necessary plugins for Git integration, job scheduling, and webhook support.
- Configure Jenkins to work with Git by setting up your Git credentials in the Jenkins Credential Manager.

Step 4: Create a Jenkins Job

- Create a new Jenkins job using the "Freestyle project" type.
- Configure the job to use a webhook trigger. You can do this by selecting the "GitHub hook trigger for GITScm polling" option in the job's settings.

Step 5: Set Up the Jenkins Job (Using Execute Shell)

- In the job configuration, go to the "Build" section.
- Add a build step of type "Execute shell."
- In the "Command" field, define the build and deployment steps using shell commands. For example:

Checkout code from Git
Build your web application (e.g., npm install, npm run build) # Copy the build artefacts to the local HTTP server directory ***rm -rf /var/www/html/webdirectory/****
cp -r * /var/www/html/webdirectory/

Step 6: Set Up a Webhook in Git Repository

- In your Git repository (e.g., on GitHub), go to "Settings" and then "Webhooks."
- Create a new webhook, and configure it to send a payload to the Jenkins webhook URL (usually http://jenkins-server/github-webhook/). Make sure to set the content type to "application/json."
- OR use “GitHub hook trigger for GITScm polling?” Under Build Trigger

Step 7: Trigger the CI/CD Pipeline

- Push changes to your Git repository. The webhook should trigger the Jenkins job automatically, executing the build and deployment steps defined in the "Execute Shell" build step.
- Monitor the Jenkins job's progress in the Jenkins web interface.

Step 8: Verify the CI/CD Pipeline

Visit the URL of your local HTTP server to verify that the web application has been updated with the latest changes.

Conclusion:

This experiment demonstrates how to set up a CI/CD pipeline for web development using Jenkins, Git, a local HTTP server, and webhooks, without the need for a Jenkinsfile. By defining and executing the build and deployment steps using the "Execute Shell" build step, you can automate your development workflow and ensure that your web application is continuously updated with the latest changes.

Exercises /Questions :

1. Explain the significance of CI/CD in the context of web development. How does it benefit the development process and end-users?
2. Describe the key components of a typical CI/CD pipeline for web development. How do Jenkins, Git, and a local HTTP server fit into this pipeline?
3. Discuss the role of version control in CI/CD. How does Git facilitate collaborative web development and CI/CD automation?
4. What is the purpose of a local HTTP server in a CI/CD workflow for web development? How does it contribute to testing and deployment?
5. Explain the concept of webhooks and their role in automating CI/CD processes. How are webhooks used to trigger Jenkins jobs in response to Git events?
6. Outline the steps involved in setting up a Jenkins job to automate CI/CD for a web application.
7. Describe the differences between Continuous Integration (CI) and Continuous Deployment (CD) in the context of web development. When might you use one without the other?
8. Discuss the advantages and challenges of using Jenkins as the automation server in a CI/CD pipeline for web development.

Experiment No. 2

Title : Exploring Git Commands through Collaborative Coding.

Objective:

The objective of this experiment is to familiarise participants with essential Git concepts and commands, enabling them to effectively use Git for version control and collaboration.

Introduction:

Git is a distributed version control system (VCS) that helps developers track changes in their codebase, collaborate with others, and manage different versions of their projects efficiently. It was created by Linus Torvalds in 2005 to address the shortcomings of existing version control systems.

Unlike traditional centralised VCS, where all changes are stored on a central server, Git follows a distributed model. Each developer has a complete copy of the repository on their local machine, including the entire history of the project. This decentralisation offers numerous advantages, such as offline work, faster operations, and enhanced collaboration.

Git is a widely used version control system that allows developers to collaborate on projects, track changes, and manage codebase history efficiently. This experiment aims to provide a hands-on introduction to Git and explore various fundamental Git commands. Participants will learn how to set up a Git repository, commit changes, manage branches, and collaborate with others using Git.

Key Concepts:

- **Repository:** A Git repository is a collection of files, folders, and their historical versions. It contains all the information about the project's history, branches, and commits.
- **Commit:** A commit is a snapshot of the changes made to the files in the repository at a specific point in time. It includes a unique identifier (SHA-1 hash), a message describing the changes, and a reference to its parent commit(s).
- **Branch:** A branch is a separate line of development within a repository. It allows developers to work on new features or bug fixes without affecting the main codebase. Branches can be merged back into the main branch when the changes are ready.
- **Merge:** Merging is the process of combining changes from one branch into another. It integrates the changes made in a feature branch into the main branch or any other target branch.
- **Pull Request:** In Git hosting platforms like GitHub, a pull request is a feature that allows developers to propose changes from one branch to another. It provides a platform for code review and collaboration before merging.

git status

- This command shows the status of your working directory, highlighting untracked files.

git add example.txt

- This stages the changes of the "example.txt" file for commit.

git commit -m "Add content to example.txt"

- This commits the staged changes with a descriptive message.

Step 3: Exploring History

Modify the content of "example.txt." Run the following commands:

git status

- Notice the modified file is shown as "modified."

git diff

- This displays the differences between the working directory and the last commit.

git log

- This displays a chronological history of commits.

Step 4: Branching and Merging

Create a new branch named "feature" and switch to it:

git branch feature

git checkout feature

or shorthand:

git checkout -b feature

- Make changes to the "example.txt" file in the "feature" branch.
- Commit the changes in the "feature" branch.
- Switch back to the "master" branch:

git checkout master

- Merge the changes from the "feature" branch into the "master" branch:

git merge feature

Step 5: Collaborating with Remote Repositories

- Create an account on a Git hosting service like GitHub (https://github.com/).
- Create a new repository on GitHub.
- Link your local repository to the remote repository:

git remote add origin <repository_url>

- Push your local commits to the remote repository:

git push origin master

Step 2: Making Changes and Creating a Branch

Navigate into the cloned repository:

cd <repository_name>

- Create a new text file named "example.txt" using a text editor.
- Add some content to the "example.txt" file.
- Save the file and return to the command line.
- Check the status of the repository:
  git status

- Stage the changes for commit:

git add example.txt

- Commit the changes with a descriptive message:
  git commit -m "Add content to example.txt"

- Create a new branch named "feature":
  git branch feature

- Switch to the "feature" branch:
  git checkout feature

Step 3: Pushing Changes to GitHub

- Add Repository URL in a variable
  git remote add origin <repository_url>
- Replace <repository_url> with the URL you copied from GitHub.
- Push the "feature" branch to GitHub:

git push origin feature

- Check your GitHub repository to confirm that the new branch "feature" is available.

Step 4: Collaborating through Pull Requests

- Create a pull request on GitHub:
- Go to the repository on GitHub.
- Click on "Pull Requests" and then "New Pull Request."
- Choose the base branch (usually "main" or "master") and the compare branch ("feature").
- Review the changes and click "Create Pull Request."
- Review and merge the pull request:
- Add a title and description for the pull request.
- Assign reviewers if needed.
- Once the pull request is approved, merge it into the base branch.

Step 5: Syncing Changes

- After the pull request is merged, update your local repository:

git checkout main

git pull origin main

Conclusion:

This experiment provided you with practical experience in performing GitHub operations using Git commands. You learned how to clone repositories, make changes, create branches, push changes to GitHub, collaborate through pull requests, and synchronise changes with remote repositories. These skills are essential for effective collaboration and version control in software development projects using Git and GitHub.

Questions:

1. Explain the difference between Git and GitHub.
2. What is a GitHub repository? How is it different from a Git repository?
3. Describe the purpose of a README.md file in a GitHub repository.
4. How do you create a new repository on GitHub? What information is required during the creation process?
5. Define what a pull request (PR) is on GitHub. How does it facilitate collaboration among developers?
6. Describe the typical workflow of creating a pull request and having it merged into the main branch.
7. How can you address and resolve merge conflicts in a pull request?
8. Explain the concept of forking a repository on GitHub. How does it differ from cloning a repository?
9. What is the purpose of creating a local clone of a repository on your machine? How is it done using Git commands?
10. Describe the role of GitHub Issues and Projects in managing a software development project. How can they be used to track tasks, bugs, and enhancements?

Benefits of Using GitLab:

- **End-to-End DevOps:** GitLab offers an integrated platform for the entire software development and delivery process, from code writing to deployment.
- **Simplicity:** GitLab provides a unified interface for version control, CI/CD, and project management, reducing the need to use multiple tools.
- **Customizability:** GitLab can be self-hosted on-premises or used through GitLab's cloud service. This flexibility allows organizations to choose the hosting option that best suits their needs.
- **Security:** GitLab places a strong emphasis on security, with features like role-based access control (RBAC), security scanning, and compliance checks.
- **Open Source and Enterprise Versions:** GitLab offers both a free, open-source Community Edition and a paid, feature-rich Enterprise Edition, making it suitable for individual developers and large enterprises alike.

Materials:

- Computer with Git installed (https://git-scm.com/downloads)
- GitLab account (https://gitlab.com/)
- Internet connection

Experiment Steps:

Step 1: Creating a Repository

- Sign in to your GitLab account.
- Click the "New" button to create a new project.
- Choose a project name, visibility level (public, private), and other settings.
- Click "Create project."

Step 2: Cloning a Repository

- Open your terminal or command prompt.
- Navigate to the directory where you want to clone the repository.
- Copy the repository URL from GitLab.
- Run the following command:
  git clone <repository_url>
- Replace <repository_url> with the URL you copied from GitLab.
- This will clone the repository to your local machine.

Step 3: Making Changes and Creating a Branch

- Navigate into the cloned repository:
  cd <repository_name>
- Create a new text file named "example.txt" using a text editor.
- Add some content to the "example.txt" file.
- Save the file and return to the command line.
- Check the status of the repository:
  git status

- Stage the changes for commit:
  git add example.txt

- Commit the changes with a descriptive message:
  git commit -m "Add content to example.txt"

- Create a new branch named "feature":
  git branch feature

- Switch to the "feature" branch:
  git checkout feature

Step 4: Pushing Changes to GitLab

- Add Repository URL in a variable
  git remote add origin <repository_url>
- Replace <repository_url> with the URL you copied from GitLab.
- Push the "feature" branch to GitLab:

git push origin feature

- Check your GitLab repository to confirm that the new branch "feature" is available.

Step 5: Collaborating through Merge Requests

1. Create a merge request on GitLab:
   - Go to the repository on GitLab.
   - Click on "Merge Requests" and then "New Merge Request."
   - Choose the source branch ("feature") and the target branch ("main" or "master").
   - Review the changes and click "Submit merge request."
2. Review and merge the merge request:
   - Add a title and description for the merge request.
   - Assign reviewers if needed.
   - Once the merge request is approved, merge it into the target branch.

Step 6: Syncing Changes

- After the merge request is merged, update your local repository:
  git checkout main
  git pull origin main

Conclusion:

This experiment provided you with practical experience in performing GitLab operations using Git commands. You learned how to create repositories, clone them to your local machine, make changes, create branches, push changes to GitLab, collaborate through merge requests, and synchronise changes with remote repositories. These skills are crucial for effective collaboration and version control in software development projects using GitLab and Git.

Questions/Exercises:

1. What is GitLab, and how does it differ from other version control platforms?
2. Explain the significance of a GitLab repository. What can a repository contain?
3. What is a merge request in GitLab? How does it facilitate the code review process?
4. Describe the steps involved in creating and submitting a merge request on GitLab.
5. What are GitLab issues, and how are they used in project management?
6. Explain the concept of a GitLab project board and its purpose in organising tasks.
7. How does GitLab address security concerns in software development? Mention some security-related features.
8. Describe the role of compliance checks in GitLab and how they contribute to maintaining software quality.

Experiment No. 5
Title: Exploring Containerization and Application Deployment with Docker

Objective:
The objective of this experiment is to provide hands-on experience with Docker containerization and application deployment by deploying an Apache web server in a Docker container. By the end of this experiment, you will understand the basics of Docker, how to create Docker containers, and how to deploy a simple web server application.

Introduction
Containerization is a technology that has revolutionised the way applications are developed, deployed, and managed in the modern IT landscape. It provides a standardised and efficient way to package, distribute, and run software applications and their dependencies in isolated environments called containers.

Containerization technology has gained immense popularity, with Docker being one of the most well-known containerization platforms. This introduction explores the fundamental concepts of containerization, its benefits, and how it differs from traditional approaches to application deployment.

Key Concepts of Containerization:

- **Containers:** Containers are lightweight, stand-alone executable packages that include everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. Containers ensure that an application runs consistently and reliably across different environments, from a developer's laptop to a production server.
- **Images:** Container images are the templates for creating containers. They are read-only and contain all the necessary files and configurations to run an application. Images are typically built from a set of instructions defined in a Dockerfile.
- **Docker:** Docker is a popular containerization platform that simplifies the creation, distribution, and management of containers. It provides tools and services for building, running, and orchestrating containers at scale.
- **Isolation:** Containers provide process and filesystem isolation, ensuring that applications and their dependencies do not interfere with each other. This isolation enhances security and allows multiple containers to run on the same host without conflicts.

Benefits of Containerization:

- **Consistency:** Containers ensure that applications run consistently across different environments, reducing the "it works on my machine" problem.
- **Portability:** Containers are portable and can be easily moved between different host machines and cloud providers.
- **Resource Efficiency:** Containers share the host operating system's kernel, which makes them lightweight and efficient in terms of resource utilization.
- **Scalability:** Containers can be quickly scaled up or down to meet changing application demands, making them ideal for microservices architectures.
- **Version Control:** Container images are versioned, enabling easy rollback to previous application states if issues arise.
- **DevOps and CI/CD:** Containerization is a fundamental technology in DevOps and CI/CD pipelines, allowing for automated testing, integration, and deployment.

Containerization vs. Virtualization:

- Containerization differs from traditional virtualization, where a hypervisor virtualizes an entire operating system (VM) to run multiple applications. In contrast:
- Containers share the host OS kernel, making them more lightweight and efficient.
- Containers start faster and use fewer resources than VMs.
- VMs encapsulate an entire OS, while containers package only the application and its dependencies.

Materials:

- A computer with Docker installed (https://docs.docker.com/get-docker/)
- A code editor
- Basic knowledge of Apache web server

Experiment Steps:

Step 1: Install Docker

- If you haven't already, install Docker on your computer by following the instructions provided on the Docker website (https://docs.docker.com/get-docker/).

Step 2: Create a Simple HTML Page

- Create a directory for your web server project.
- Inside this directory, create a file named index.html with a simple "Hello, Docker!" message. This will be the content served by your Apache web server.

Step 3: Create a Dockerfile

- Create a Dockerfile in the same directory as your web server project. The Dockerfile defines how your Apache web server application will be packaged into

a Docker container. Here's an
- example:

Dockerfile

 # Use an official Apache image as the base image
FROM httpd:2.4

 # Copy your custom HTML page to the web server's document root
COPY index.html /usr/local/apache2/htdocs/

Step 4: Build the Docker Image

- Build the Docker image by running the following command in the same directory as your Dockerfile:
  docker build -t my-apache-server .
- Replace my-apache-server with a suitable name for your image.

Step 5: Run the Docker Container

Start a Docker container from the image you built:
docker run -p 8080:80 -d my-apache-server

- This command maps port 80 in the container to port 8080 on your host machine and runs the container in detached mode.

Step 6: Access Your Apache Web Server

Access your Apache web server by opening a web browser and navigating to http://localhost:8080. You should see the "Hello, Docker!" message served by your Apache web server running within the Docker container.

Step 7: Cleanup

Stop the running Docker container:
docker stop <container_id>

- Replace <container_id> with the actual ID of your running container.
- Optionally, remove the container and the Docker image:

docker rm <container_id>
docker rmi my-apache-server

Conclusion:

In this experiment, you explored containerization and application deployment with Docker by deploying an Apache web server in a Docker container. You learned how to create a Dockerfile, build a Docker image, run a Docker container, and access your web server application from your host machine. Docker's containerization capabilities make it a valuable tool for packaging and deploying applications consistently across different environments.

Experiment No. 6
Title: Applying CI/CD Principles to Web Development Using Jenkins, Git, using Docker Containers

Objective:
The objective of this experiment is to set up a CI/CD pipeline for a web application using Jenkins, Git, Docker containers, and GitHub webhooks. The pipeline will automatically build, test, and deploy the web application whenever changes are pushed to the Git repository, without the need for a pipeline script.

Introduction:
Continuous Integration and Continuous Deployment (CI/CD) principles are integral to modern web development practices, allowing for the automation of code integration, testing, and deployment. This experiment demonstrates how to implement CI/CD for web development using Jenkins, Git, Docker containers, and GitHub webhooks without a pipeline script. Instead, we'll utilize Jenkins' "GitHub hook trigger for GITScm polling" feature.

In the fast-paced world of modern web development, the ability to deliver high-quality software efficiently and reliably is paramount. Continuous Integration and Continuous Deployment (CI/CD) are integral principles and practices that have revolutionized the way software is developed, tested, and deployed. These practices bring automation, consistency, and speed to the software development lifecycle, enabling development teams to deliver code changes to production with confidence.

Continuous Integration (CI):
CI is the practice of frequently and automatically integrating code changes from multiple contributors into a shared repository. The core idea is that developers regularly merge their code into a central repository, triggering automated builds and tests. Key aspects of CI include:

- Automation: CI tools, like Jenkins, Travis CI, or CircleCI, automate the building and testing of code whenever changes are pushed to the repository.
- Frequent Integration: Developers commit and integrate their code changes multiple times a day, reducing integration conflicts and catching bugs early.
- Testing: Automated tests, including unit tests and integration tests, are run to ensure that new code changes do not introduce regressions.
- Quick Feedback: CI provides rapid feedback to developers about the quality and correctness of their code changes.

Continuous Deployment (CD):
CD is the natural extension of CI. It is the practice of automatically and continuously deploying code changes to production or staging environments after successful integration and testing. Key aspects of CD include:

- Automation: CD pipelines automate the deployment process, reducing the risk of human error and ensuring consistent deployments.
- Deployment to Staging: Code changes are deployed first to a staging environment where further testing and validation occur.
- Deployment to Production: After passing all tests in the staging environment, code changes are automatically deployed to the production environment, often with zero downtime.
- Rollbacks: In case of issues, CD pipelines provide the ability to rollback to a previous version quickly.

Benefits of CI/CD in Web Development:

- Rapid Development: CI/CD accelerates development cycles by automating time-consuming tasks, allowing developers to focus on coding.
- Quality Assurance: Automated testing ensures code quality, reducing the number of bugs and regressions.
- Consistency: CI/CD ensures that code is built, tested, and deployed consistently, regardless of the development environment.
- Continuous Feedback: Developers receive immediate feedback on the impact of their changes, improving collaboration and productivity.
- Reduced Risk: Automated deployments reduce the likelihood of deployment errors and downtime, enhancing reliability.
- Scalability: CI/CD can scale to accommodate projects of all sizes, from small startups to large enterprises.

Materials:

- A computer with Docker installed (https://docs.docker.com/get-docker/)
- Jenkins installed and configured (https://www.jenkins.io/download/)
- A web application code repository hosted on GitHub

Experiment Steps:

Step 1: Set Up the Web Application and Git Repository

- Create a simple web application or use an existing one. Ensure it can be hosted in a Docker container.

- Initialise a Git repository for your web application and push it to GitHub.

Step 2: Install and Configure Jenkins

- Install Jenkins on your computer or server following the instructions for your operating system (https://www.jenkins.io/download/).
- Open Jenkins in your web browser (usually at http://localhost:8080) and complete the initial setup, including setting up an admin user and installing necessary plugins.
- Configure Jenkins to work with Git by setting up Git credentials in the Jenkins Credential Manager.

Step 3: Create a Jenkins Job

- Create a new Jenkins job using the "Freestyle project" type.
- In the job configuration, specify a name for your job and choose "This project is parameterized."
- Add a "String Parameter" named GIT_REPO_URL and set its default value to your Git repository URL.
- Set Branches to build -> Branch Specifier to the working Git branch (ex */master)
- In the job configuration, go to the "Build Triggers" section and select the "GitHub hook trigger for GITScm polling" option. This enables Jenkins to listen for GitHub webhook triggers.

Step 4: Configure Build Steps

- In the job configuration, go to the "Build" section.
- Add build steps to execute Docker commands for building and deploying the containerized web application. Use the following commands:

```
# Remove the existing container if it exists
docker rm --force container1

# Build a new Docker image
docker build -t nginx-image1 .

# Run the Docker container
docker run -d -p 8081:80 --name=container1 nginx-image1
```

- These commands remove the existing container (if any), build a Docker image named "nginx-image1," and run a Docker container named "container1" on port

8081.

Step 5: Set Up a GitHub Webhook

- In your GitHub repository, navigate to "Settings" and then "Webhooks."
- Create a new webhook, and configure it to send a payload to the Jenkins webhook URL (usually http://jenkins-server/github-webhook/). Set the content type to "application/json."

Step 6: Trigger the CI/CD Pipeline

- Push changes to your GitHub repository. The webhook will trigger the Jenkins job automatically, executing the build and deployment steps defined in the job configuration.
- Monitor the Jenkins job's progress in the Jenkins web interface.

Step 7: Verify the Deployment

Access your web application by opening a web browser and navigating to http://localhost:8081 (or the appropriate URL if hosted elsewhere).

Conclusion:

This experiment demonstrates how to apply CI/CD principles to web development using Jenkins, Git, Docker containers, and GitHub webhooks. By configuring Jenkins to listen for GitHub webhook triggers and executing Docker commands in response to code changes, you can automate the build and deployment of your web application, ensuring a more efficient and reliable development workflow.

Exercise / Questions :

1. Explain the core principles of Continuous Integration (CI) and Continuous Deployment (CD) in the context of web development. How do these practices enhance the software development lifecycle?
2. Discuss the key differences between Continuous Integration and Continuous Deployment. When might you choose to implement one over the other in a web development project?
3. Describe the role of automation in CI/CD. How do CI/CD pipelines automate code integration, testing, and deployment processes?
4. Explain the concept of a CI/CD pipeline in web development. What are the typical stages or steps in a CI/CD pipeline, and why are they important?
5. Discuss the benefits of CI/CD for web development teams. How does CI/CD impact the speed, quality, and reliability of software delivery?
6. What role do version control systems like Git play in CI/CD workflows for web development? How does version control contribute to collaboration and automation?
7. Examine the challenges and potential risks associated with implementing CI/CD in web development. How can these challenges be mitigated?
8. Provide examples of popular CI/CD tools and platforms used in web development.

Experiment Steps:

Step 1: Create a GitHub Account

- Visit the GitHub website (https://github.com/).
- Click on the "Sign Up" button and follow the instructions to create your GitHub account.

Step 2: Create a Sample GitHub Repository

- Log in to your GitHub account.
- Click the "+" icon in the top-right corner and select "New Repository."
- Give your repository a name (e.g., "my-web-pages") and provide an optional description.
- Choose the repository visibility (public or private).
- Click the "Create repository" button.

Step 3: Set Up a Google Cloud Platform Project

- Log in to your Google Cloud Platform account.
- Create a new GCP project by clicking on the project drop-down in the GCP Console (https://console.cloud.google.com/).
- Click on "New Project" and follow the prompts to create a project.

Step 4: Connect GitHub to Google Cloud Build

- In your GCP Console, navigate to "Cloud Build" under the "Tools" section.
- Click on "Triggers" in the left sidebar.
- Click the "Connect Repository" button.
- Select "GitHub (Cloud Build GitHub App)" as the source provider.
- Authorise Google Cloud Build to access your GitHub account.
- Choose your GitHub repository ("my-web-pages" in this case) and branch.
- Click "Create."

Step 5: Create a CI/CD Configuration File

- In your GitHub repository, create a configuration file named **cloudbuild.yaml**. This file defines the CI/CD pipeline steps.
- Add a simple example configuration to copy web pages to an Apache web server.

Here's an example:

steps:
-name: 'gcr.io/cloud-builders/gsutil'
args: ['-m', 'rsync', '-r', 'web-pages/', 'gs://your-bucket-name']

- Replace 'gs://your-bucket-name' with the actual Google Cloud Storage bucket where your Apache web server serves web pages.
- Commit and push this file to your GitHub repository.

Step 6: Trigger the CI/CD Pipeline

- Make changes to your web pages or configuration.
- Push the changes to your GitHub repository.
- Go to your GCP Console and navigate to "Cloud Build" > "Triggers."
- You should see an automatic trigger for your repository. Click the trigger to see details.
- Click "Run Trigger" to manually trigger the CI/CD pipeline.

Step 7: Monitor the CI/CD Pipeline

- In the GCP Console, navigate to "Cloud Build" to monitor the progress of your build and deployment.
- Once the pipeline is complete, your web pages will be copied to the specified Google Cloud Storage bucket.

Step 8: Access Your Deployed Web Pages

- Configure your Apache web server to serve web pages from the Google Cloud Storage bucket.
- Access your deployed web pages by visiting the appropriate URL.

Conclusion:

In this experiment, you created a GitHub account, set up a basic CI/CD pipeline on Google Cloud Platform, and deployed web pages to an Apache web server. This demonstrates how CI/CD can automate the deployment of web content, making it easier to manage and update web applications efficiently.

Exercise / Questions:

1. What is the primary purpose of Continuous Integration and Continuous Deployment (CI/CD) in software development, and how does it benefit development teams using GitHub, GCP, and AWS?
2. Explain the role of GitHub in a CI/CD pipeline. How does GitHub facilitate version control and collaboration in software development?
3. What are the key services and offerings provided by Google Cloud Platform (GCP) that are commonly used in CI/CD pipelines, and how do they contribute to the automation and deployment of applications?
4. Similarly, describe the essential services and tools offered by Amazon Web Services (AWS) that are typically integrated into a CI/CD workflow.
5. Walk through the basic steps of a CI/CD pipeline from code development to production deployment, highlighting the responsibilities of each stage.
6. How does Continuous Integration (CI) differ from Continuous Deployment (CD)? Explain how GitHub Actions or a similar CI tool can be configured to build and test code automatically.
7. In the context of CI/CD, what is a staging environment, and why is it important in

- State Management: Terraform maintains a state file that keeps track of the real-world resources it manages. This state file helps Terraform understand the current state of the infrastructure and determine what changes are needed to align it with the desired configuration.
- Plan and Apply: Terraform provides commands to plan and apply changes to your infrastructure. The "plan" command previews changes before applying them, ensuring you understand the impact.
- Dependency Management: Terraform automatically handles resource dependencies. If one resource relies on another, Terraform determines the order of provisioning.
- Modularity: Terraform configurations can be organized into modules, allowing you to create reusable and shareable components.
- Community and Ecosystem: Terraform has a large and active community, contributing modules, providers, and best practices. The Terraform Registry hosts a wealth of pre-built modules and configurations.

Typical Workflow:

- Configuration Definition: Define your infrastructure configuration using Terraform's declarative syntax. Describe the resources, providers, and dependencies in your *.tf files.
- Initialization: Run terraform init to initialize your Terraform project. This command downloads required providers and sets up your working directory.
- Planning: Execute terraform plan to create an execution plan. Terraform analyzes your configuration and displays what changes will be made to the infrastructure.
- Provisioning: Use terraform apply to apply the changes and provision resources. Terraform will create, update, or delete resources as needed to align with your configuration.
- State Management: Terraform maintains a state file (by default, terraform.tfstate) that tracks the current state of the infrastructure.
- Modifications: As your infrastructure requirements change, update your Terraform configuration files and run terraform apply again to apply the changes incrementally.
- Destruction: When resources are no longer needed, you can use terraform destroy to remove them. Be cautious, as this action can't always be undone.

Advantages of Terraform:

- Predictable and Repeatable: Terraform configurations are repeatable and idempotent. The same configuration produces the same results consistently.
- Collaboration: Infrastructure configurations can be versioned, shared, and collaborated on by teams, promoting consistency.
- Multi-Cloud: Terraform's multi-cloud support allows you to manage infrastructure across different cloud providers with the same tool.
- Community and Modules: A rich ecosystem of modules, contributed by the community, accelerates infrastructure provisioning.

- Terraform has become a fundamental tool in the DevOps and infrastructure automation landscape, enabling organizations to manage infrastructure efficiently and with a high degree of control.

Materials:

- A computer with Terraform installed (https://www.terraform.io/downloads.html)
- Access to a cloud provider (e.g., AWS, Google Cloud, Azure) with appropriate credentials configured

Experiment Steps:

Step 1: Install and Configure Terraform

- Download and install Terraform on your local machine by following the instructions for your operating system (https://www.terraform.io/downloads.html).

  Verify the installation by running:
  terraform version

- Configure your cloud provider's credentials using environment variables or a configuration file. For example, if you're using AWS, you can configure your AWS access and secret keys as environment variables:
  export AWS_ACCESS_KEY_ID=your_access_key
  export AWS_SECRET_ACCESS_KEY=your_secret_key

Step 2: Create a Terraform Configuration File

Create a new directory for your Terraform project:
mkdir my-terraform-project cd my-terraform-project

Inside the project directory, create a Terraform configuration file named main.tf. This file will define your infrastructure resources. For a simple example, let's create an AWS S3 bucket:

```
provider "aws" {
  region = "us-east-1" # Change to your desired region
}

resource "aws_s3_bucket" "example_bucket" {
  bucket = "my-unique-bucket-name" # Replace with a globally unique name
  acl    = "private"
}
```

Step 3: Initialize and Apply the Configuration

- Initialize the Terraform working directory to download the necessary provider plugins:
  terraform init
- Validate the configuration to ensure there are no syntax errors:
  terraform validate

- Apply the configuration to create the AWS S3 bucket:
  terraform apply

- Terraform will display a summary of the planned changes. Type "yes" when prompted to apply the changes.

Step 4: Verify the Infrastructure

After the Terraform apply command completes, you can verify the created infrastructure. For an S3 bucket, you can check the AWS Management Console or use the AWS CLI:
aws s3 ls

- You should see your newly created S3 bucket.

Step 5: Modify and Destroy Infrastructure

- To modify your infrastructure, you can edit the main.tf file and then re-run terraform apply. For example, you can add new resources or update existing ones.
    - To destroy the infrastructure when it's no longer needed, run:

terraform destroy
Confirm the destruction by typing "yes."

Explanation:

- In this experiment, you created a simple Terraform configuration to provision an AWS EC2 instance.
- The main.tf file defines an AWS provider, specifying the desired region, and a resource block that defines an EC2 instance with the specified Amazon Machine Image (AMI) and instance type.
- Running terraform init initializes the Terraform project, and terraform plan provides a preview of the changes Terraform will make.
    - terraform applies the changes, creating the AWS EC2 instance.
    - To clean up resources, terraform destroy can be used.

Conclusion: