INDEX

List of Practical

Experiment No. 1

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.

- **Remote Repository:** A remote repository is a copy of the Git repository stored on a server, enabling collaboration among multiple developers. It can be hosted on platforms like GitHub, GitLab, or Bitbucket.

Basic Git Commands:

- **git init**: Initialises a new Git repository in the current directory.
- **git clone**: Creates a copy of a remote repository on your local machine.
- **git add**: Stages changes for commit, preparing them to be included in the next commit.
- **git commit**: Creates a new commit with the staged changes and a descriptive message.
- **git status**: Shows the current status of the working directory, including tracked and untracked files.
- **git log**: Displays a chronological list of commits in the repository, showing their commit messages, authors, and timestamps.
- **git branch**: Lists, creates, or deletes branches within the repository.
- **git checkout**: Switches between branches, commits, or tags. It's used to navigate through the repository's history.
- **git merge**: Combines changes from different branches, integrating them into the current branch.
- **git pull**: Fetches changes from a remote repository and merges them into the current branch.
- **git push**: Sends local commits to a remote repository, updating it with the latest changes.

Materials:

- Computer with Git installed (https://git-scm.com/downloads)
- Command-line interface (Terminal, Command Prompt, or Git Bash)

Experiment Steps:

Step 1: Setting Up Git Repository

- Open the command-line interface on your computer.
- Navigate to the directory where you want to create your Git repository.
- Run the following commands:

git init

- This initialises a new Git repository in the current directory.

Step 2: Creating and Committing Changes

- Create a new text file named "example.txt" using any text editor.
- Add some content to the "example.txt" file.
- In the command-line interface, run the following commands:

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

Conclusion:

Through this experiment, participants gained a foundational understanding of Git's essential commands and concepts. They learned how to set up a Git repository, manage changes, explore commit history, create and merge branches, and collaborate with remote repositories. This knowledge equips them with the skills needed to effectively use Git for version control and collaborative software development.

Exercise:

1. Explain what version control is and why it is important in software development. Provide examples of version control systems other than Git.
2. Describe the typical workflow when working with Git, including initialising a repository, committing changes, and pushing to a remote repository. Use a real-world example to illustrate the process.
3. Discuss the purpose of branching in Git and how it facilitates collaborative development. Explain the steps involved in creating a new branch, making changes, and merging it back into the main branch.
4. What are merge conflicts in Git, and how can they be resolved? Provide a step-by-step guide on how to handle a merge conflict.
5. Explain the concept of remote repositories in Git and how they enable collaboration among team members. Describe the differences between cloning a repository and adding a remote.
6. Discuss different branching strategies, such as feature branching and Gitflow. Explain the advantages and use cases of each strategy.
7. Describe various Git commands and techniques for undoing changes, such as reverting commits, resetting branches, and discarding uncommitted changes.
8. What are Git hooks, and how can they be used to automate tasks and enforce coding standards in a Git repository? Provide examples of practical use cases for Git hooks.
9. List and explain five best practices for effective and efficient Git usage in a collaborative software development environment.
10. Discuss security considerations in Git, including how to protect sensitive information like passwords and API keys. Explain the concept of Git signing and why it's important.

Experiment No. 2
Title: Implement GitHub Operations using Git.

Objective:
The objective of this experiment is to guide you through the process of using Git commands to interact with GitHub, from cloning a repository to collaborating with others through pull requests.

Introduction:
GitHub is a web-based platform that offers version control and collaboration services for software development projects. It provides a way for developers to work together, manage code, track changes, and collaborate on projects efficiently. GitHub is built on top of the Git version control system, which allows for distributed and decentralised development.

Key Features of GitHub:

- **Version Control:** GitHub uses Git, a distributed version control system, to track changes to source code over time. This allows developers to collaborate on projects while maintaining a history of changes and versions.
- **Repositories**: A repository (or repo) is a collection of files, folders, and the entire history of a project. Repositories on GitHub serve as the central place where code and project-related assets are stored.
- **Collaboration:** GitHub provides tools for team collaboration. Developers can work together on the same project, propose changes, review code, and discuss issues within the context of the project.
- **Pull Requests:** Pull requests (PRs) are proposals for changes to a repository. They allow developers to submit their changes for review, discuss the changes, and collaboratively improve the code before merging it into the main codebase.
- **Issues and Projects:** GitHub allows users to track and manage project-related issues, enhancements, and bugs. Projects and boards help organize tasks, track progress, and manage workflows.
- **Forks and Clones:** Developers can create copies (forks) of repositories to work on their own versions of a project. Cloning a repository allows developers to create a local copy of the project on their machine.
- **Branching and Merging:** GitHub supports branching, where developers can create separate lines of development for features or bug fixes. Changes made in branches can be merged back into the main codebase.
- **Actions and Workflows:** GitHub Actions enable developers to automate workflows, such as building, testing, and deploying applications, based on triggers like code pushes or pull requests.
- **GitHub Pages:** This feature allows users to publish web content directly from a GitHub repository, making it easy to create websites and documentation for projects.

Benefits of Using GitHub:

- **Collaboration:** GitHub facilitates collaborative development by providing tools for code review, commenting, and discussion on changes.
- **Version Control:** Git's version control features help track changes, revert to previous versions, and manage different branches of development.

- **Open Source:** GitHub is widely used for hosting open-source projects, making it easier for developers to contribute and for users to find and use software.
- **Community Engagement:** GitHub fosters a community around projects, enabling interaction between project maintainers and contributors.
- **Code Quality:** The code review process on GitHub helps maintain code quality and encourages best practices.
- **Documentation:** GitHub provides a platform to host project documentation and wikis, making it easier to document codebases and project processes.
- **Continuous Integration/Continuous Deployment (CI/CD):** GitHub Actions allows for automating testing, building, and deploying applications, streamlining the development process.

Materials:

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

Experiment Steps:

Step 1: Cloning a Repository

- Sign in to your GitHub account.
- Find a repository to clone (you can use a repository of your own or any public repository).
- Click the "Code" button and copy the repository URL.
- Open your terminal or command prompt.
- Navigate to the directory where you want to clone the repository.
- Run the following command:

git clone <repository_url>

- Replace <repository_url> with the URL you copied from GitHub.
- This will clone the repository to your local machine.

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?

Experiment No. 3

Title: Implement GitLab Operations using Git.

Objective:
The objective of this experiment is to guide you through the process of using Git commands to interact with GitLab, from creating a repository to collaborating with others through merge requests.

Introduction:
GitLab is a web-based platform that offers a complete DevOps lifecycle toolset, including version control, continuous integration/continuous deployment (CI/CD), project management, code review, and collaboration features. It provides a centralized place for software development teams to work together efficiently and manage the entire development process in a single platform.

Key Features of GitLab:

- **Version Control:** GitLab provides version control capabilities using Git, allowing developers to track changes to source code over time. This enables collaboration, change tracking, and code history maintenance.
- **Repositories:** Repositories on GitLab are collections of files, code, documentation, and assets related to a project. Each repository can have multiple branches and tags, allowing developers to work on different features simultaneously.
- **Continuous Integration/Continuous Deployment (CI/CD):** GitLab offers robust CI/CD capabilities. It automates the building, testing, and deployment of code changes, ensuring that software is delivered rapidly and reliably.
- **Merge Requests:** Merge requests in GitLab are similar to pull requests in other platforms. They enable developers to propose code changes, collaborate, and get code reviewed before merging it into the main codebase.
- **Issues and Project Management:** GitLab includes tools for managing project tasks, bugs, and enhancements. Issues can be assigned, labeled, and tracked, while project boards help visualize and manage work.
- **Container Registry:** GitLab includes a container registry that allows users to store and manage Docker images for applications.
- **Code Review and Collaboration:** Built-in code review tools facilitate collaboration among team members. Inline comments, code discussions, and code snippets are integral parts of this process.
- **Wiki and Documentation:** GitLab provides a space for creating project wikis and documentation, ensuring that project information is easily accessible and well-documented.
- **Security and Compliance:** GitLab offers security scanning, code analysis, and compliance features to help identify and address security vulnerabilities and ensure code meets compliance standards.
- **GitLab Pages:** Similar to GitHub Pages, GitLab Pages lets users publish static websites directly from a GitLab repository.

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. 4
Title: Implement BitBucket Operations using Git.

Objective:
The objective of this experiment is to guide you through the process of using Git commands to interact with Bitbucket, from creating a repository to collaborating with others through pull requests.

Introduction:
Bitbucket is a web-based platform designed to provide version control, source code management, and collaboration tools for software development projects. It is widely used by teams and individuals to track changes in code, collaborate on projects, and streamline the development process. Bitbucket offers Git and Mercurial as version control systems and provides features to support code collaboration, continuous integration/continuous deployment (CI/CD), and project management.

Key Features of Bitbucket:

- Version Control: Bitbucket supports both Git and Mercurial version control systems, allowing developers to track changes, manage code history, and work collaboratively on projects.
- Repositories: In Bitbucket, a repository is a container for code, documentation, and other project assets. It houses different branches, tags, and commits that represent different versions of the project.
- Collaboration: Bitbucket enables team collaboration through features like pull requests, code reviews, inline commenting, and team permissions. These tools help streamline the process of merging code changes.
- Pull Requests: Pull requests in Bitbucket allow developers to propose and review code changes before they are merged into the main codebase. This process helps ensure code quality and encourages collaboration.
- Code Review: Bitbucket provides tools for efficient code review, allowing team members to comment on specific lines of code and discuss changes within the context of the code itself.
- Continuous Integration/Continuous Deployment (CI/CD): Bitbucket integrates with CI/CD pipelines, automating processes such as building, testing, and deploying code changes to various environments.
- Project Management: Bitbucket offers project boards and issue tracking to help manage tasks, track progress, and plan project milestones effectively.
- Bitbucket Pipelines: This feature allows teams to define and automate CI/CD pipelines directly within Bitbucket, ensuring code quality and rapid delivery.
- Access Control and Permissions: Bitbucket allows administrators to define user roles, permissions, and access control settings to ensure the security of repositories and project assets.

Benefits of Using Bitbucket:

- Version Control: Bitbucket's integration with Git and Mercurial provides efficient version control and code history tracking.

- Collaboration: The platform's collaboration tools, including pull requests and code reviews, improve code quality and facilitate team interaction.
- CI/CD Integration: Bitbucket's integration with CI/CD pipelines automates testing and deployment, resulting in faster and more reliable software delivery.
- Project Management: Bitbucket's project management features help teams organize tasks, track progress, and manage milestones.
- Flexibility: Bitbucket offers both cloud-based and self-hosted options, providing flexibility to choose the deployment method that suits the organization's needs.
- Integration: Bitbucket integrates with various third-party tools, services, and extensions, enhancing its functionality and extending its capabilities.

Materials:

- Computer with Git installed (https://git-scm.com/downloads)
- Bitbucket account (https://bitbucket.org/)
- Internet connection

Experiment Steps:

Step 1: Creating a Repository

- Sign in to your Bitbucket account.
- Click the "Create" button to create a new repository.
- Choose a repository name, visibility (public or private), and other settings.
- Click "Create repository."

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 Bitbucket.
- Run the following command:

git clone <repository_url>

- Replace <repository_url> with the URL you copied from Bitbucket.
- 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 Bitbucket

- Add Repository URL in a variable

git remote add origin <repository_url>

- Replace <repository_url> with the URL you copied from Bitbucket.
- Push the "feature" branch to Bitbucket:

git push origin feature

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

Step 5: Collaborating through Pull Requests

1. Create a pull request on Bitbucket:
    - Go to the repository on Bitbucket.
    - Click on "Create pull request."
    - Choose the source branch ("feature") and the target branch ("main" or "master").
    - Review the changes and click "Create pull request."
2. 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 target branch.

Step 6: 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 Bitbucket operations using Git commands. You learned how to create repositories, clone them to your local machine, make changes, create branches, push changes to Bitbucket, 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 Bitbucket and Git.

Questions/Exercises:

Q.1 What is Bitbucket, and how does it fit into the DevOps landscape?
Q.2 Explain the concept of branching in Bitbucket and its significance in collaborative development.
Q.3 What are pull requests in Bitbucket, and how do they facilitate code review and collaboration?

Q.4 How can you integrate code quality analysis and security scanning tools into Bitbucket's CI/CD pipelines?
Q.5 What are merge strategies in Bitbucket, and how do they affect the merging process during pull requests?

Experiment No. 5
Title: Applying CI/CD Principles to Web Development Using Jenkins, Git, and Local HTTP Server

Objective:
The objective of this experiment is to set up a CI/CD pipeline for a web development project using Jenkins, Git, and webhooks, without the need for a Jenkinsfile. You will learn how to automatically build and deploy a web application to a local HTTP server whenever changes are pushed to the Git repository, using Jenkins' "Execute Shell" build step.

Introduction:
Continuous Integration and Continuous Deployment (CI/CD) is a critical practice in modern software development, allowing teams to automate the building, testing, and deployment of applications. This process ensures that software updates are consistently and reliably delivered to end-users, leading to improved development efficiency and product quality.

In this context, this introduction sets the stage for an exploration of how to apply CI/CD principles specifically to web development using Jenkins, Git, and a local HTTP server. We will discuss the key components and concepts involved in this process.

Key Components:

- **Jenkins:** Jenkins is a widely used open-source automation server that helps automate various aspects of the software development process. It is known for its flexibility and extensibility and can be employed to create CI/CD pipelines.
- **Git:** Git is a distributed version control system used to manage and track changes in source code. It plays a crucial role in CI/CD by allowing developers to collaborate, track changes, and trigger automation processes when code changes are pushed to a repository.
- **Local HTTP Server:** A local HTTP server is used to host and serve web applications during development. It is where your web application can be tested before being deployed to production servers.

CI/CD Principles:

- **Continuous Integration (CI):** CI focuses on automating the process of integrating code changes into a shared repository frequently. It involves building and testing the application each time code is pushed to the repository to identify and address issues early in the development cycle.
- **Continuous Deployment (CD):** CD is the practice of automatically deploying code changes to production or staging environments after successful testing. CD aims to minimize manual intervention and reduce the time between code development and its availability to end-users.

The CI/CD Workflow:

- **Code Changes:** Developers make changes to the web application's source code locally.

- **Git Repository:** Developers push their code changes to a Git repository, such as GitHub or Bitbucket.
- **Webhook:** A webhook is configured in the Git repository to notify Jenkins whenever changes are pushed.
- **Jenkins Job:** Jenkins is set up to listen for webhook triggers. When a trigger occurs, Jenkins initiates a CI/CD pipeline.
- **Build and Test:** Jenkins executes a series of predefined steps, which may include building the application, running tests, and generating artifacts.
- **Deployment:** If all previous steps are successful, Jenkins deploys the application to a local HTTP server for testing.
- **Verification:** The deployed application is tested locally to ensure it functions as expected.
- **Optional Staging:** For more complex setups, there might be a staging environment where the application undergoes further testing before reaching production.
- **Production Deployment:** If the application passes all tests, it can be deployed to the production server.

Benefits of CI/CD in Web Development:

- **Rapid Development:** CI/CD streamlines development processes, reducing manual tasks and allowing developers to focus on writing code.
- **Improved Quality:** Automated testing helps catch bugs early, ensuring higher code quality.
- **Faster Time to Market:** Automated deployments reduce the time it takes to release new features or updates.
- **Consistency:** The process ensures that code is built, tested, and deployed consistently, reducing the risk of errors.

Materials:

- A computer with administrative privileges
- Jenkins installed and running (https://www.jenkins.io/download/)
- Git installed (https://git-scm.com/downloads)
- A local HTTP server for hosting web content (e.g., Apache, Nginx)
- A Git repository (e.g., on GitHub or Bitbucket)

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.
9. Explain how a Jenkinsfile is typically used in a Jenkins-based CI/CD pipeline. What are the benefits of defining pipeline stages in code?
10. Provide examples of test cases that can be automated as part of a CI/CD process for web development. How does automated testing contribute to code quality and reliability in web applications?

Experiment No. 6
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.

Exercise/Questions:

1. Explain the concept of containerization. How does it differ from traditional virtualization methods?
2. Discuss the key components of a container. What are images and containers in the context of containerization?
3. What is Docker, and how does it contribute to containerization? Explain the role of Docker in building, running, and managing containers.
4. Describe the benefits of containerization for application deployment and management. Provide examples of scenarios where containerization is advantageous.
5. Explain the concept of isolation in containerization. How do containers provide process and filesystem isolation for applications?

6. Discuss the importance of container orchestration tools such as Kubernetes in managing containerized applications. What problems do they solve, and how do they work?
7. Compare and contrast containerization platforms like Docker, containerd, and rkt. What are their respective strengths and weaknesses?
8. Explain the process of creating a Docker image. What is a Dockerfile, and how does it help in image creation?
9. Discuss the security considerations in containerization. What measures can be taken to ensure the security of containerized applications?
10. Explore real-world use cases of containerization in software development and deployment. Provide examples of industries or companies that have benefited from containerization technologies.

Experiment No. 7
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. How do these tools facilitate the implementation of CI/CD principles?
9. Explain the concept of "Infrastructure as Code" (IaC) and its relevance to CI/CD. How can IaC be used to automate infrastructure provisioning in web development projects?
10. Discuss the cultural and organisational changes that may be necessary when adopting CI/CD practices in a web development team. How does CI/CD align with DevOps principles and culture?

Experiment No. 8
Title: Demonstrate Maven Build Life Cycle

Objective:
The objective of this experiment is to gain hands-on experience with the Maven build lifecycle by creating a simple Java project and executing various Maven build phases.

Introduction:
Maven is a widely-used build automation and project management tool in the Java ecosystem. It provides a clear and standardised build lifecycle for Java projects, allowing developers to perform various tasks such as compiling code, running tests, packaging applications, and deploying artefacts. This experiment aims to demonstrate the Maven build lifecycle and its different phases.

Key Maven Concepts:

- Project Object Model (POM): The POM is an XML file named pom.xml that defines a project's configuration, dependencies, plugins, and goals. It serves as the project's blueprint and is at the core of Maven's functionality.
- Build Lifecycle: Maven follows a predefined sequence of phases and goals organized into build lifecycles. These lifecycles include clean, validate, compile, test, package, install, and deploy, among others.
- Plugin: Plugins are extensions that provide specific functionality to Maven. They enable tasks like compiling code, running tests, packaging artifacts, and deploying applications.
- Dependency Management: Maven simplifies dependency management by allowing developers to declare project dependencies in the POM file. Maven downloads these dependencies from repositories like Maven Central.
- Repository: A repository is a collection of artifacts (compiled libraries, JARs, etc.) that Maven uses to manage dependencies. Maven Central is a popular public repository, and organisations often maintain private repositories.

Maven Build Life Cycle:
The Maven build process is organised into a set of build lifecycles, each comprising a sequence of phases. Here are the key build lifecycles and their associated phases:

Clean Lifecycle:
- clean: Deletes the target directory, removing all build artifacts.

Default Lifecycle:
- validate: Validates the project's structure.
- compile: Compiles the project's source code.
- test: Runs tests using a suitable testing framework.
- package: Packages the compiled code into a distributable format (e.g., JAR, WAR).
- verify: Runs checks on the package to verify its correctness.
- install: Installs the package to the local repository.
- deploy: Copies the final package to a remote repository for sharing.

Site Lifecycle:

- site: Generates project documentation.

Each phase within a lifecycle is executed in sequence, and the build progresses from one phase to the next. Developers can customise build behaviour by configuring plugins and goals in the POM file.

Materials:

- A computer with Maven installed (https://maven.apache.org/download.cgi)
- A code editor (e.g., Visual Studio Code, IntelliJ IDEA)
- Java Development Kit (JDK) installed (https://www.oracle.com/java/technologies/javase-downloads.html)

Experiment Steps:

Step 1: Setup Maven and Java

- Ensure that you have Maven and JDK installed on your system. You can verify their installations by running the following commands:

mvn -v
java -version

Step 2: Create a Maven Java Project

- Create a new directory for your project, e.g., MavenDemo.
- Inside the project directory, create a simple Java class, e.g., HelloWorld.java, with a main method that prints "Hello, Maven!".

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, Maven!");
   }
}
```

Create a pom.xml file (Maven Project Object Model) in the project directory. This file defines project metadata, dependencies, and build configurations. Here's a minimal example:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                                    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.example</groupId>
   <artifactId>MavenDemo</artifactId>
   <version>1.0-SNAPSHOT</version>
</project>
```

Step 3: Explore the Maven Build Phases

- Maven has several build phases, and each phase is responsible for specific tasks. In this step, we'll explore some of the most commonly used build phases.

- Clean Phase: To clean the project (remove generated files), run:

mvn clean

- Compile Phase: To compile the Java source code, run:

mvn compile

- Test Phase: To execute unit tests, run:

mvn test

- Package Phase: To package the application into a JAR file, run:

mvn package

- Install Phase: To install the project artifacts (e.g., JAR) into your local Maven repository, run:

mvn install

- Deploy Phase: To deploy artifacts to a remote Maven repository, configure your pom.xml and run:

mvn deploy

Step 4: Run the Application

- After running the mvn package command, you can find the generated JAR file (e.g., MavenDemo-1.0-SNAPSHOT.jar) in the target directory. Run the application using:

java -cp target/MavenDemo-1.0-SNAPSHOT.jar HelloWorld

Conclusion:

This experiment demonstrates the Maven build lifecycle by creating a simple Java project and executing various Maven build phases. Maven simplifies the build process by providing a standardized way to manage dependencies, compile code, run tests, and package applications. Understanding these build phases is essential for Java developers using Maven in their projects.

Exercise/Questions:

1. What is Maven, and why is it commonly used in software development?
2. Explain the purpose of the pom.xml file in a Maven project.
3. How does Maven simplify dependency management in software projects?
4. What are Maven plugins, and how do they enhance the functionality of Maven?
5. List the key phases in the Maven build lifecycle, and briefly describe what each phase does.
6. What is the primary function of the clean phase in the Maven build lifecycle?
7. In Maven, what does the compile phase do, and when is it typically executed?

8. How does Maven differentiate between the test and verify phases in the build lifecycle?
9. What is the role of the install phase in the Maven build lifecycle, and why is it useful?
10. Explain the difference between a local repository and a remote repository in the context of Maven.

Experiment No. 9
Title: Demonstrating Container Orchestration using Kubernetes

Objective:
The objective of this experiment is to introduce students to container orchestration using Kubernetes and demonstrate how to deploy a containerized web application. By the end of this experiment, students will have a basic understanding of Kubernetes concepts and how to use Kubernetes to manage containers.

Introduction:
Container orchestration is a critical component in modern application deployment, allowing you to manage, scale, and maintain containerized applications efficiently. Kubernetes is a popular container orchestration platform that automates many tasks associated with deploying, scaling, and managing containerized applications. This experiment will demonstrate basic container orchestration using Kubernetes by deploying a simple web application.

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Developed by Google and later donated to the Cloud Native Computing Foundation (CNCF), Kubernetes has become the de facto standard for container orchestration in modern cloud-native application development.

Key Concepts in Kubernetes:

- Containerization: Kubernetes relies on containers as the fundamental unit for packaging and running applications. Containers encapsulate an application and its dependencies, ensuring consistency across various environments.
- Cluster: A Kubernetes cluster is a set of machines, known as nodes, that collectively run containerized applications. A cluster typically consists of a master node (for control and management) and multiple worker nodes (for running containers).
- Nodes: Nodes are individual machines (virtual or physical) that form part of a Kubernetes cluster. Nodes run containerized workloads and communicate with the master node to manage and orchestrate containers.
- Pod: A pod is the smallest deployable unit in Kubernetes. It can contain one or more tightly coupled containers that share the same network and storage namespace. Containers within a pod are typically used to run closely related processes.
- Deployment: A Deployment is a Kubernetes resource that defines how to create, update, and scale instances of an application. It ensures that a specified number of replicas are running at all times.
- Service: A Service is an abstraction that exposes a set of pods as a network service. It provides a stable IP address and DNS name for accessing the pods, enabling load balancing and discovery.
- Namespace: Kubernetes supports multiple virtual clusters within the same physical cluster, called namespaces. Namespaces help isolate resources and provide a scope for organizing and managing workloads.

Key Features of Kubernetes:

- Automated Scaling: Kubernetes can automatically scale the number of replicas of an application based on resource usage or defined metrics. This ensures applications can handle varying workloads efficiently.
- Load Balancing: Services in Kubernetes can distribute traffic among pods, providing high availability and distributing workloads evenly.
- Self-healing: Kubernetes monitors the health of pods and can automatically restart or replace failed instances to maintain desired application availability.
- Rolling Updates and Rollbacks: Kubernetes allows for controlled, rolling updates of applications, ensuring zero-downtime deployments. If issues arise, rollbacks can be performed with ease.
- Storage Orchestration: Kubernetes provides mechanisms for attaching storage volumes to containers, enabling data persistence and sharing.
- Configuration Management: Kubernetes supports configuration management through ConfigMaps and Secrets, making it easy to manage application configurations.
- Extensibility: Kubernetes is highly extensible, with a vast ecosystem of plugins and extensions, including Helm charts for packaging applications and custom resources for defining custom objects.

Kubernetes has become a cornerstone of cloud-native application development, enabling organisations to build, deploy, and scale containerized applications effectively. Its ability to abstract away infrastructure complexities, ensure application reliability, and provide consistent scaling makes it a powerful tool for modern software development and operations.

Materials:

- A computer with Kubernetes installed (https://kubernetes.io/docs/setup/)
- Docker installed (https://docs.docker.com/get-docker/)

Experiment Steps:

Step 1: Create a Dockerized Web Application

- Create a simple web application (e.g., a static HTML page) or use an existing one.
- Create a Dockerfile to package the web application into a Docker container. Here's an example Dockerfile for a simple web server:

```
# Use an official Nginx base image
FROM nginx:latest
# Copy the web application files to the Nginx document root
COPY ./webapp /usr/share/nginx/html
```

- Build the Docker image:

```
docker build -t my-web-app .
```

Step 2: Deploy the Web Application with Kubernetes
Create a Kubernetes Deployment YAML file (web-app-deployment.yaml) to deploy the web application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-web-app-deployment
spec:
  replicas: 3               # Number of pods to create
  selector:
    matchLabels:
      app: my-web-app       # Label to match pods
  template:
    metadata:
      labels:
        app: my-web-app     # Label assigned to pods
    spec:
      containers:
      - name: my-web-app-container
        image: my-web-app:latest  # Docker image to use
        ports:
        - containerPort: 80     # Port to expose
```

Explanation of web-app-deployment.yaml:

- apiVersion: Specifies the Kubernetes API version being used (apps/v1 for Deployments).
- kind: Defines the type of resource we're creating (a Deployment in this case).
- metadata: Contains metadata for the Deployment, including its name.
- spec: Defines the desired state of the Deployment.
- replicas: Specifies the desired number of identical pods to run. In this example, we want three replicas of our web application.
- selector: Specifies how to select which pods are part of this Deployment. Pods with the label app: my-web-app will be managed by this Deployment.
- template: Defines the pod template for the Deployment.
- metadata: Contains metadata for the pods created by this template.
- labels: Assigns the label app: my-web-app to the pods created by this template.
- spec: Specifies the configuration of the pods.
- containers: Defines the containers to run within the pods. In this case, we have one container named my-web-app-container using the my-web-app:latest Docker image.
- ports: Specifies the ports to expose within the container. Here, we're exposing port 80.

Step 3: Deploy the Application

- Apply the deployment configuration to your Kubernetes cluster:

kubectl apply -f web-app-deployment.yaml

Step 4: Verify the Deployment

- Check the status of your pods:

kubectl get pods

Conclusion:

In this experiment, you learned how to create a Kubernetes Deployment for container orchestration. The web-app-deployment.yaml file defines the desired state of the application, including the number of replicas, labels, and the Docker image to use. Kubernetes automates the deployment and scaling of the application, making it a powerful tool for managing containerized workloads.

Questions/Exercises:

1. Explain the core concepts of Kubernetes, including pods, nodes, clusters, and deployments. How do these concepts work together to manage containerized applications?
2. Discuss the advantages of containerization and how Kubernetes enhances the orchestration and management of containers in modern application development.
3. What is a Kubernetes Deployment, and how does it ensure high availability and scalability of applications? Provide an example of deploying a simple application using a Kubernetes Deployment.
4. Explain the purpose and benefits of Kubernetes Services. How do Kubernetes Services facilitate load balancing and service discovery within a cluster?
5. Describe how Kubernetes achieves self-healing for applications running in pods. What mechanisms does it use to detect and recover from pod failures?
6. How does Kubernetes handle rolling updates and rollbacks of applications without causing downtime? Provide steps to perform a rolling update of a Kubernetes application.
7. Discuss the concept of Kubernetes namespaces and their use cases. How can namespaces be used to isolate and organize resources within a cluster?
8. Explain the role of Kubernetes ConfigMaps and Secrets in managing application configurations. Provide examples of when and how to use them.
9. What is the role of storage orchestration in Kubernetes, and how does it enable data persistence and sharing for containerized applications?
10. Explore the extensibility of Kubernetes. Describe Helm charts and custom resources, and explain how they can be used to customize and extend Kubernetes functionality.

Experiment No. 10
Title: Create the GitHub Account to demonstrate CI/CD pipeline using Cloud Platform.

Objective:
The objective of this experiment is to help you create a GitHub account and set up a basic CI/CD pipeline on GCP. You will learn how to connect your GitHub repository to GCP, configure CI/CD using Cloud Build, and automatically deploy web pages to an Apache web server when code is pushed to your repository.

Introduction:
Continuous Integration and Continuous Deployment (CI/CD) pipelines are essential for automating the deployment of web applications. In this experiment, we will guide you through creating a GitHub account and setting up a basic CI/CD pipeline using Google Cloud Platform (GCP) to copy web pages for an Apache HTTP web application.

Continuous Integration and Continuous Deployment (CI/CD) is a crucial practice in modern software development. It involves automating the processes of code integration, testing, and deployment to ensure that software changes are consistently and reliably delivered to production. GitHub, Google Cloud Platform (GCP), and Amazon Web Services (AWS) are popular tools and platforms that, when combined, enable a powerful CI/CD pipeline.

Key Components:

- GitHub: GitHub is a web-based platform for version control and collaboration. It allows developers to host and manage their source code repositories, track changes, collaborate with others, and automate workflows.
- Google Cloud Platform (GCP): GCP is a cloud computing platform that provides a wide range of cloud services, including computing, storage, databases, machine learning, and more. It can be used to host applications and services.
- Amazon Web Services (AWS): AWS is another cloud computing platform that offers a comprehensive set of cloud services. It is often used for hosting infrastructure, containers, databases, and more.

Basic CI/CD Workflow:

A basic CI/CD workflow using GitHub, GCP, and AWS typically includes the following steps:

1. **Code Development:** Developers work on code changes and commit them to a GitHub repository.
2. **Continuous Integration (CI):**
    a. GitHub Actions or a CI tool like Jenkins is used to automatically build, test, and package the application whenever code changes are pushed to the repository.
    b. Automated tests are executed to ensure code quality.
3. **Continuous Deployment (CD):**
    a. Once code changes pass CI, the application can be automatically deployed to a staging environment.
    b. Integration and acceptance tests are performed in the staging environment.

4. **Deployment to Production:**
    a. If all tests in the staging environment pass, the application can be automatically deployed to the production environment in GCP or AWS.
5. **Monitoring and Logging:**
    a. Monitoring tools are used to track the application's performance and detect issues.
    b. Logging and analytics tools are used to gain insights into application behavior.
6. **Feedback Loop:**
    a. Any issues or failures detected in production are reported back to the development team for further improvements.
    b. The cycle repeats as new code changes are developed and deployed.

Benefits of Basic CI/CD with GitHub, GCP, and AWS:

1. **Automation:** CI/CD automates repetitive tasks, reducing the risk of human error and speeding up the delivery process.
2. **Consistency:** CI/CD ensures that all code changes go through the same testing and deployment processes, leading to consistent and reliable results.
3. **Faster Time to Market:** Automated deployments enable faster delivery of new features and bug fixes to users.
4. **Improved Collaboration:** GitHub's collaboration features enable teams to work together seamlessly, and CI/CD pipelines keep everyone on the same page.
5. **Scalability:** Cloud platforms like GCP and AWS provide scalable infrastructure to handle varying workloads.
6. **Efficiency:** Developers can focus on writing code while CI/CD pipelines take care of building, testing, and deploying applications.

Materials:

- A computer with internet access
- A Google Cloud Platform account (https://cloud.google.com/)
- A GitHub account (https://github.com/)

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.yam**l. 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 the deployment process? How does it differ from a production environment?
8. What are the primary benefits of using automation for deployment in a CI/CD pipeline, and how does this automation contribute to consistency and reliability in software releases?
9. Discuss the significance of monitoring, logging, and feedback loops in a CI/CD workflow. How do these components help in maintaining and improving application quality and performance?
10. In terms of scalability and flexibility, explain how cloud platforms like GCP and AWS enhance the CI/CD process, especially when dealing with variable workloads and resource demands.

Experiment 11 (Content Beyond Syllabus)
Title: Demonstrating Infrastructure as Code (IaC) with Terraform

Objective:
The objective of this experiment is to introduce you to Terraform and demonstrate how to create, modify, and destroy infrastructure resources locally using Terraform's configuration files and commands.

Introduction:
Terraform is a powerful Infrastructure as Code (IaC) tool that allows you to define and provision infrastructure using a declarative configuration language. In this experiment, we will demonstrate how to use Terraform on your local machine to create and manage infrastructure resources in a cloud environment.

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It enables the creation, management, and provisioning of infrastructure resources and services across various cloud providers, on-premises environments, and third-party services. Terraform follows a declarative syntax and is designed to make infrastructure provisioning predictable, repeatable, and automated.

Key Concepts and Features:

- Declarative Syntax: Terraform uses a declarative configuration language (HCL - HashiCorp Configuration Language) to define infrastructure as code. Instead of specifying step-by-step instructions, you declare what resources you want, and Terraform figures out how to create and manage them.
- Infrastructure as Code (IaC): Terraform treats infrastructure as code, allowing you to version, share, and collaborate on infrastructure configurations just like you would with application code.
- Providers: Terraform supports a wide range of providers, including AWS, Azure, Google Cloud, and more. Each provider allows you to manage resources specific to that platform.
- Resources: Resources are the individual infrastructure components you define in your Terraform configuration. Examples include virtual machines, databases, networks, and security groups.
- 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:
In this experiment, you learned how to use Terraform on your local machine to create and manage infrastructure resources as code. Terraform simplifies infrastructure provisioning, modification, and destruction by providing a declarative way to define and maintain your infrastructure, making it a valuable tool for DevOps and cloud engineers.

Exercises / Questions

1. What is Terraform, and what is its primary purpose in the context of infrastructure management and automation?
2. Explain the key difference between declarative and imperative programming approaches. How does Terraform use a declarative approach in its configuration files?
3. What is the role of Terraform providers, and how do they enable the management of resources across various cloud platforms and services?
4. Describe the significance of the Terraform state file. How does it help Terraform keep track of the current state of the infrastructure?

5. In Terraform, what are resources, and how are they defined in configuration files? Provide examples of common resources.
6. What is the purpose of Terraform modules, and how do they facilitate code reusability and modularity in infrastructure configurations?
7. Explain the steps involved in a typical Terraform workflow, from defining infrastructure as code to applying changes to the infrastructure.
8. How does Terraform handle resource dependencies and ensure the correct order of resource provisioning within a configuration?
9. What are the benefits of using Terraform's plan and apply commands? How do they help prevent unintended changes to the infrastructure?
10. Discuss Terraform's support for multi-cloud deployments. How can Terraform be used to manage resources across different cloud providers within the same configuration?