Task 1: Git Basic - Clone and Commit Changes

- **Solution**:
    1. Clone the repository:

       ```
       git clone https://github.com/username/repository.git
       cd repository
       ```

    2. Make changes to any file in the repository (e.g., edit `README.md`).
    3. Stage the changes:

       ```
       git add README.md
       ```

    4. Commit the changes:

       ```
       git commit -m "Updated README file"
       ```

    5. Push the changes:

       ```
       git push origin main
       ```

- **Answer**:
    - `git commit -m "message"` creates a snapshot of your changes, which are saved in the local repository. The commit message describes the purpose or content of the changes made.

Task 2: Git Branching and Merging

- **Solution**:
    1. Create a new branch:

       ```
       git checkout -b feature-branch
       ```

    2. Modify a file (e.g., add new content to `README.md`).
    3. Stage and commit the changes:

       ```
       git add README.md
       git commit -m "Added feature details"
       ```

    4. Switch back to the `main` branch:

       ```
       git checkout main
       ```

5. Merge the `feature-branch` into `main`:

   ```
   git merge feature-branch
   ```

6. Push the changes to the remote repository:

   ```
   git push origin main
   ```

- **Answer**:
  - `git merge` combines changes from different branches into the current branch. It merges the commit histories of both branches.

Task 3: Git Rebase vs. Git Merge

- **Solution**:
  1. Create a new branch and commit changes:

     ```
     git checkout -b feature-branch
     echo "Some feature changes" > feature.txt
     git add feature.txt
     git commit -m "Added feature changes"
     ```

  2. Merge the branch:

     ```
     git checkout main
     git merge feature-branch
     ```

  3. Rebase the branch:

     ```
     git checkout feature-branch
     git rebase main
     ```

  4. Compare the commit histories:

     ```
     git log --oneline
     ```

- **Answer**:
  - `git merge` combines the branches, preserving both histories. In contrast, `git rebase` rewrites commit history, making it appear as though the changes were applied directly on top of the target branch, resulting in a cleaner, linear history.

Task 4: Git Stash to Save Changes Temporarily

- **Solution**:
    1. Modify some files but do not commit (e.g., edit `feature.txt`).
    2. Stash the changes:

       ```
       git stash
       ```

    3. Switch to another branch:

       ```
       git checkout main
       ```

    4. Return to the original branch and apply the stashed changes:

       ```
       git checkout feature-branch
       git stash pop
       ```

- **Answer**:
    - `git stash` temporarily saves your changes, allowing you to switch branches without committing them. You can later reapply the changes using `git stash pop`.

Task 5: CI/CD - Set Up a Simple CI Pipeline with GitLab CI

- **Solution**:
    1. Create a new GitLab repository.
    2. Add the `.gitlab-ci.yml` file to the root of the repository with the following content:

       ```
       stages:
         - build

       build_job:
         stage: build
         script:
           - echo "Hello, GitLab CI!"
       ```

    3. Commit and push the file:

       ```
       git add .gitlab-ci.yml
       git commit -m "Add GitLab CI configuration"
       git push origin main
       ```

    4. In GitLab, go to **CI/CD > Pipelines** to see the pipeline running.

- **Answer**:
  - `.gitlab-ci.yml` defines the pipeline stages and jobs. The `build` job runs a simple script to print "Hello, GitLab CI!". This file automates your CI process by defining the steps to execute when code is pushed to the repository.

Task 6: Continuous Delivery (CD) with Jenkins

- **Solution**:
  1. Install Jenkins and start it on your local machine.
  2. Create a new Jenkins pipeline job.
  3. Configure the pipeline to pull from your Git repository (e.g., using a GitHub webhook).
  4. Add the build and deploy steps:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying to staging server...'
            }
        }
    }
}
```

  5. Trigger the pipeline manually or automatically after a commit.
- **Answer**:
  - Jenkins automates the build and deployment process. After configuring the pipeline, Jenkins automatically builds and deploys the application to the staging server whenever changes are committed to the repository.

Task 7: Setting Up Nginx as a Reverse Proxy

- **Solution**:
    1. Install Nginx on your server:

       ```
       sudo apt update
       sudo apt install nginx
       ```

    2. Configure Nginx as a reverse proxy by editing `/etc/nginx/sites-available/default`:

       ```
       nginx
       Copy code
       server {
           listen 80;
           server_name example.com;

           location / {
               proxy_pass http://localhost:3000;
               proxy_set_header Host $host;
               proxy_set_header X-Real-IP $remote_addr;
               proxy_set_header X-Forwarded-For
       $proxy_add_x_forwarded_for;
           }
       }
       ```

    3. Restart Nginx:

       ```
       sudo systemctl restart nginx
       ```

    4. Test by accessing `http://example.com` in a browser, which should forward requests to the backend server.
- **Answer**:
    - The `proxy_pass` directive forwards requests to the backend server. The `proxy_set_header` directives ensure that relevant information (such as the original IP address) is passed along to the backend.

Task 8: Load Balancing with Nginx

- **Solution**:
    1. Set up two or more backend servers (e.g., `localhost:3000` and `localhost:3001`).
    2. Configure the `upstream` directive in the Nginx config:

       ```
       upstream backend {
           server localhost:3000;
           server localhost:3001;
       }
       ```

```
server {
    listen 80;
    location / {
        proxy_pass http://backend;
    }
}
```

3. Restart Nginx:

```
sudo systemctl restart nginx
```

4. Send requests to `http://example.com` and Nginx will distribute traffic between the two backend servers.

- **Answer**:
  - Nginx uses round-robin by default for load balancing, distributing traffic evenly across multiple backend servers. You can also configure other algorithms like IP hash or least_conn.

Task 9: Apache HTTPD Reverse Proxy Setup

- **Solution**:
  1. Install Apache HTTPD:

```
sudo apt install apache2
```

  2. Enable necessary modules:

```
sudo a2enmod proxy proxy_http
```

  3. Configure reverse proxy in the Apache config file (`/etc/apache2/sites-available/000-default.conf`):

```
<VirtualHost *:80>
    ServerName example.com
    ProxyPass / http://localhost:3000/
    ProxyPassReverse / http://localhost:3000/
</VirtualHost>
```

4. Restart Apache:

```
sudo systemctl restart apache2
```

5. Test by accessing `http://example.com`.

- **Answer**:
  - Apache uses `ProxyPass` and `ProxyPassReverse` to forward client requests to a backend server. `ProxyPass` forwards requests, and `ProxyPassReverse` ensures the correct response headers are returned to the client.

Task 10: Continuous Deployment with Docker and Jenkins

- **Solution**:
  1. Create a simple Dockerized application (e.g., a Node.js app):

```
# Dockerfile
FROM node:14
WORKDIR /app
COPY . /app
RUN npm install
CMD ["node", "app.js"]
```

  2. Build the Docker image:

```
docker build -t myapp .
```

  3. Set up Jenkins to monitor the Git repository.
  4. Configure the Jenkins pipeline to build and deploy the Docker container:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script {
                    docker.build("myapp")
                }
            }
        }
        stage('Deploy') {
            steps {
                script {
                    docker.run("myapp")
                }
            }
        }
    }
}
```

5. Trigger the Jenkins job and monitor the deployment process.

- **Answer**:
  - Docker containers provide a consistent environment for application deployment. Jenkins automates building and deploying the Docker container, ensuring the latest changes are deployed efficiently to the production server.