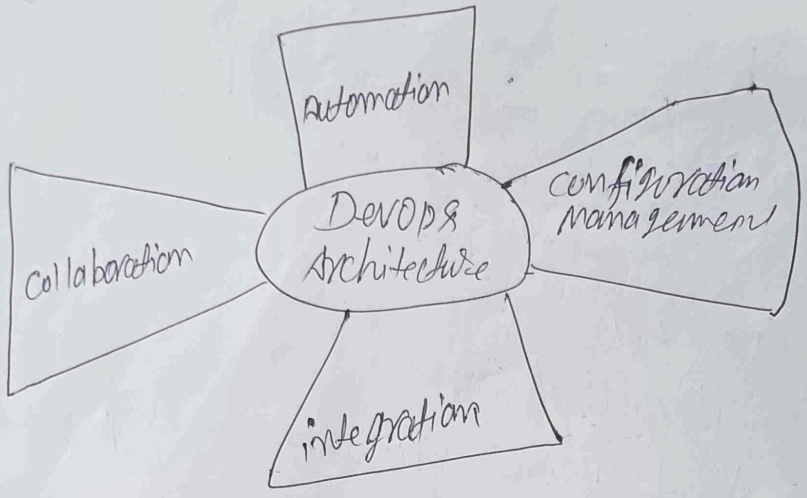
DeV+Ops

* DevOps: The DevOps is the combination of two words one is development and second is operations. it is a culture to ~~devete~~ promote the development and operation process collectively.

various DevOps tools such as
(i) Git (ii) Github (iii) Docker (iv) Ansible
(v) puppet (vi) Jenkins (vii) chef (viii) Nagios
(ix) kubernates

DevOps is combination of two word one is software development and second is operation. this allows a single team to handle the entire application life cycle from development to testing, deployment and operations. DevOps helps to reduce disconnection between software developer, quality assurance engineer and system administrator. DevOps promote collaboration b/w development & operation team to deploy code to production faster in an automated and repeatable way.

* DevOps Architecture features
Here are some key features of
(i)

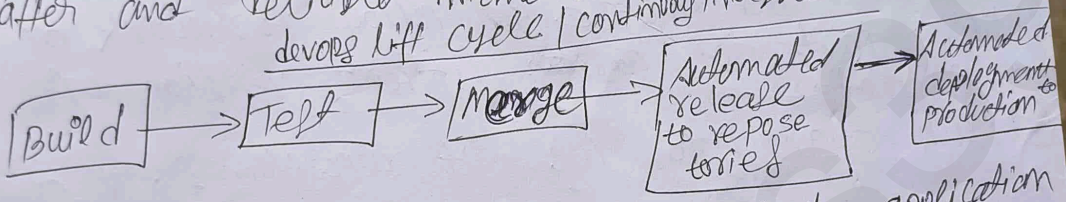


(i) Automation: Automation can reduce time consumption specially during the testing and deployment phase. the productivity increase and release are made quicker by automation, this will lead in catching bugs quickly so that it can be fixed easily for continigious develing each code is ~~designed for~~ define through automated ~~test~~, cloud based services and builds this promotes production using automated deploys.

(ii) Collaboration: The development and operational team collaborate as a devops team which improve cultural model of a team become more productive with their productivity which strengthens accountibily & ownership ~~the~~. the team share their responsibility and work closely and effectively which can in-term makes the deployment ~~to~~ o. production faster.

Integration: Applications need to be integrated with other components in the environment, the Integration phase is where the existing code is combined with new functionality and tested. Continuous and testing enables continuous development the frequency in the release and microservice lead to significant operational challenges to overcome such problems Continuous integration and develop are implemented to delivery in quicker, safer and reliable manner.

devops life cycle | continuous integration



(iv) Configuration Management: It ensures the application to interact with only those resources that are connected with the environment in which it runs external configuration to application is separated from source code. The configuration files are not created where during deployment or can be loaded at runtime depending on environment on which its running.

Intro to git

Git is a popular version control system and it is used to (i) tracking code changes (ii) tracking who made changes (iii) coding collaboration

* what does git do?
(i) manage the projects with their repositories
(ii) clone a projects to work on a local copy.
(iii) control and track changes with staging and commiting.
(iv) Branch and merge to allow for work on different parts of and different version of a project.
(v) push changes to the git repositories.
(vi) pull local updates to the main project.

* Working with Git.
(i) Initilize a git on a folder and making it a repository.
(ii) git now created a hiddem folder to keep track of changes in that folder.
(iii) when a file is changed, added or deleted it is considered modify
(iv) you select modifies modified files you want to stage.
(v) The staged files are committed which prompt git to store a permament snapshot of the file.
(vi) Git allows you to see the full history of every commit you can revert back to any previous commit.

⑥ git does not store a seprate copy of every file in every commit but keeps track of changes made in each commit.

why Git ?

(i) Over 70% of developers use git (ii) developers can work together from any where.

(iii) developers can see full history of project.

(iv) developers can revert to earlier version of the project

Github: Github is not same as git, github makes the tool that uses git. github is the largest host of source code in the world has been owned by microsoft since 2018.

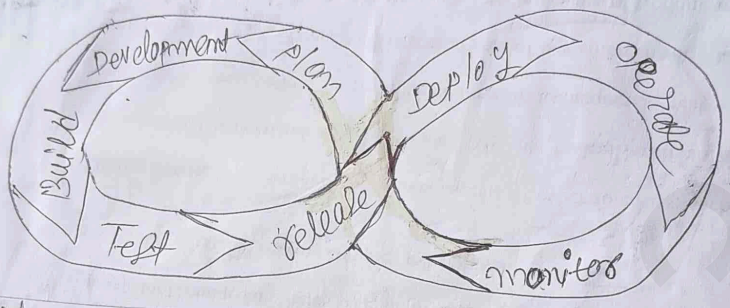
* diff blw

git	github
(i) It is a software	(i) It is a service.
(ii) It is installed locally in the system.	(ii) It is hosted on web.
(iii) git is a command line tool	(iii) github provides a graphical interface.
(iv) It is a tool to manage different version of edits made to file in a git repo.	It is a space to upload a copy of the git repository.
(v) It provides functionalities like version control system, source code management.	It provides functionality of git like VCS, source code management as well as adding few of its own features.

(pg) Name various devops tools and phases.

→

* DevOps Lifecycle : The DevOps lifecycle refers to a series of practices and process that bring development (Dev) and operations (Ops) teams together to improve collaboration, automate workflows, and deliver high-quality software explanation of the the DevOps lifecycle.



① plan : Teams discuss and plan what need to be developed, considering customer requirements and feedback. Tools like Jira or Trello are often used.

② Development : Developers write code and test the code for the software. This stage emphasizes collaboration, using tools like Git for version control.

③ Build : The written code is compiled and packaged into a deployable format (like a container or executable). Automation tools like Jenkins or Maven are used here,

④ Test : The software is strictly tested to find and fix bugs. Automated testing tools like Selenium or JUnit ensure the code works as expected.

⑤ Release : After successful testing, the software is prepared for deployment. This involves finalizing configurations and approvals.

⑥ Deploy: The software is deployed to production environments. Deployment automation tools like Kubernetes or Docker ensure seamless rollouts.

⑦ Operate: The software runs in the production environment, serving users. Teams monitor its performance and ensure it functions without issues.

⑧ Monitor: The system is continuously monitored for performance, errors or downtime. Tools like Prometheus or Nagios are used to gather and analyze data.

PYQ Functionality of DevOps

(i) Collaboration & communication: DevOps removes barrier b/w developers and operational teams everyone works together.

(ii) continuous Integration (CI): Developers regularly merge code into shared repo. Automatic build, test are performed Automatically.

(iii) continuous Delivery /development: Software is automatically developed to production after passing test.

(iv) Automation: DevOps automates code builds, testing, deployment using various devops tools like Jenkins, docker, kubernetes.

(v) Monitoring & Feedback: Devops tools like Grafana, AWS cloudwatch monitor system performance and give real-time feedback.

(vi) Infrastructure as code (IaC): Servers, networks, and databases are managed using code. Ensures consistent environment (Terraform).

10/2/25 SL A+

Assignment - 1

Q1. what is the Git command to discard all changes and reset to the last commit, also discuss some significant git commands in details.

ans⇒ Git command to Discard All changes and Reset to the Last Commit.

To discard all local changes and reset the repository to the last committed state. use the following command.

```
git reset --hard HEAD
```

- git reset → Resets the current branch to the specified state.
- --hard: Discards all local changes in working directory.
- HEAD: Refers to the latest commit.

If you also want to remove untracked files:

```
git clean -df
```

-d: Removes untracked directories
-f: forces the deletion of untracked file.

Significant Git Commands and Their Details.

① Initializing a git Repository in the current directory:

```
git init
```

② Coloning a repository of a remote repository.

```
git clone <repository-url>
```

③ checking Repository status shows the state of the working directory and staged changes.

```
git status
```

④ Adding changes to staging Area (Stage file for Commit):

```
git add <file>   # Adds a specific file
git add .        # Adds all changes.
```

⑤ Committing changes (keeps snapshot of the changes to the repository).

`git commit -m "commit message"`

⑥ viewing commit History (shows the commit history):

`git log`

⑦ Branch management:

`git branch <branch_name>` # creates a new branch

`git checkout <branch_name>` #switches to a branch

`git checkout -b <branch_name>` #create and switches to a new branch

⑧ Merging Branches (merges the specified branch into the current branch).

`git merge <branch_name>`

⑨ pushing changes to remote (pushing local commit to the remote repo)

`git push origin <branch_name>`

⑩ pulling changes from Remote (fetches and merges changes from a remote branch)

`git pull origin <branch_name>`

⑪ Resetting to a specific commit:

`git reset --hard <commit_hash>`

⑫ stashing changes (saving uncommitted work)

`git stash`

`git stash pop` # Applies the last stashed changes.

`git stash list` # Lists stashed changes.

(13) Removing a file from Git but keeping it locally.

`git rm --cached <file>`

(14) undoing the last commit (Before push)

`git reset --soft HEAD~1`

(15) viewing Remote Repository URLs.

`git remote -v`

(16) Tagging a commit with a version number.

`git tag -a v1.0 -m "version 1.0"`

`git push origin v1.0`

(17) viewing differences in files shows difference b/w working directory and last commit.

`git diff`

(18) Fetching changes from Remote (Retrieves changes from remote repo without merging).

`git fetch origin`

(19) Configuration and Setup (Set up user information):

`git config --global user.name "Raufhan"`

`git config --global user.email "your email"`

(20) List Remote repositories:

`git remote -v`

(21) Rename a Remote repository:

`git remote rename origin <new-origin>`

(22) Remove a Remote Repository:

`git remote remove <remote-name>`

Q. Discuss the cha... microservice ar... How do you ad... any ⇒ Micros... a large ... in...

Q2. Discuss the challenges of implementing microservices architecture in a DevOps environment. How do you address these challenges?

ans ⇒ Microservices is an architectural approach where a large application is broken down into smaller, independent services, each responsible for a specific function. These services communicate with each other through APIs, often using REST or messaging protocols. Microservices enable scalability, flexibility, and faster development cycles compared to monolithic architecture. Challenges of implementing microservices in a DevOps Environment.

(i) Increased Complexity: Managing multiple services instead of a single monolithic application makes development, monitoring, and debugging more complex.

(ii) Service communication & latency: Services communicate over a network, which introduces latency and potential failures especially with synchronous calls. that can lead to higher latency.

(iii) Deployment & CI/CD management: microservices require independent deployment pipelines, which increases the complexity of continuous integration / continuous Deployment (CI/CD).

(iv) Monitoring & Logging: Traditional monitoring tools may not be sufficient for distributed systems. collecting and correlating logs from multiple services is difficult.

(v) Security concerns: More services mean more API endpoints, increasing the attack surface. Authentication and authorization need to be managed across services.

⑥ Data management & Consistency: Each microservice may have its own database, leading to challenges in maintaining consistency.

⑦ Team collaboration & ownership: Different teams managing different services can lead to inconsistencies in technology choices and dependencies. Defining clear ownership of services is critical.

- How to Address These challenges

① Use containerization & orchestration:
Solution: Use Docker for packaging microservices and Kubernetes for managing their deployment.

② Implement API Gateway & service Mesh:
Solution: Use an API Gateway (eg. AWS API Gateway) to manage service requests and a service mesh (eg. Linkerd) for secure communication.

③ Automate CI/CD pipelines:
Solution: Use tools like Jenkins, Github Actions, or GitLab CI/CD to automate deployments.

④ Centralized Logging & monitoring:
Solution: Use tools like ELK stack (Elasticsearch, Logstash, Kibana), prometheus, and Grafana for monitoring and logging.

⑤ Strengthen Security measures:
Solution: Implement OAuth, JWT tokens, and Role-Based Access control. Use security tool like Vault for secrets management.

Q3 Explain the concept & its benefits in ... differ from ... Ans ⇒ Immutable ... where ... virtual ...

Q3. Explain the concept of "Immutable Infrastructure" & its benefits in DevOps environment. How does it differ from traditional Infrastructure management?

ans ⇒ Immutable infrastructure is a DevOps practice where infrastructure components (servers, containers, virtual machines) are never modified after deployment. Instead of updating or patching running systems, you can create and deploy a new instance with the updated configuration, while the old instance is destroyed. This approach is commonly implemented using tools like Docker, Kubernetes, Terraform.

Benefits of Immutable Infrastructure.

(i) Consistency & Reliability: Ensures all environments (dev, staging, production) remain identical, reducing configuration drift.

(ii) Easy Rollbacks: Since infrastructure is version-controlled and deployed as new instances, rolling back to a previous version is straightforward.

(iii) Security & Compliance: Eliminates security vulnerabilities that arise from outdated software since new instances are built with the latest patches.

(iv) Scalability & Automation: Works well with auto-scaling and container orchestration tools like Kubernetes, improving deployment speed.

(v) Reduced Debugging & Maintenance Effort: No need to manually update servers, reducing human errors.

Immutable vs Traditional Infrastructure

Features	Immutable infrast	Traditional infrastruc
Modification	Never modified after deployment	Regular updates, patches and changes applied
Updates	Deploy new instance	update existing systems manually.
Configuration ~~drift~~ drift	No ~~dr~~ drift.	Risk of drift due to manual changes
Rollback	Easily ~~to~~ revert.	difficult to revert.
Security	More secure.	less secure.

Q4. Explain the importance of monitoring & observability in a DevOps Ecosystem. Discuss key metrics & tools used in monitoring distributed system & microservices architecture.

Sol^n: Importance of monitoring & observability in a DevOps Ecosystem.

(i) Proactive Issue Detection: Helps identify and resolve issues before they impact users.

(ii) Improved Performance & Reliability: Ensures systems run efficiently and meet service level Agreements.

(iii) Faster incident Response: Reduces Mean Time to Detect (MTTD) and Mean Time to Resolve (MTTR).

(iv) Enhance Security & compliance: Identifies vulnerabilities and ensures compliance with industry regulations.

(v) Better User Experience: Ensures uptime and responsiveness of applications.

Key Metrics For Monitoring Distributed System & Microservices Architecture.

(1.) Infrastructure Metrics

- CPU Usage: Measures CPU consumption across services.
- Memory Usage: Tracks RAM usage to prevent memory services.
- Disk I/O & storage: Ensures sufficient storage capacity.
- Network Latency & Throughput: Monitors data transfer speed and network efficiency.

(2.) Application Metrics:

- Request Latency: Time taken to process a request.
- Error Rates: percentage of failed requests or errors.
- Throughput (TPS/QPS): Transactions or queries per second.
- Response Time: Measures end-to-end request handling time.

(3.) Microservices-Specific Metrics:

- Service Availability: Ensures service is Operational.
- API Gateway Metrics: Monitors API calls, errors, and response times.

(4.) Log & Tracing Metrics:

- Distributed Tracing: Tracks requests across microservices
- Log Analysis: Helps debug failure using structured logs.

(5.) Business Metrics

- Conversion Rates: Measure effectiveness of transaction.
- User Engagement: Tracks active users and interactions

Key Tools for Monitoring & Observability

(i) Prometheus: An open-source time-series database used for collecting and querying metrics.

(ii) Grafana: A visualization tool that can create dashboards and alerts based on metrics from various sources.

(iii) Elasticsearch: A search and analytics engine used for storing and analyzing logs.

(iv) Kibana: A visualization tool that can create dashboards and visualization based on Elasticsearch data.

(v) Zipkin: Another open-source distributed tracing system ~~used for monitoring and troubleshooting microservices.~~

Q5. Describe Concept of "GitOps" & its role in modern DevOps practices.

ans ⇒ GitOps is a DevOps methodology that uses Git as the single source of truth for defining and managing infrastructure and application deployment. it automates infrastructure provisioning, configuration management and application deployment using declarative Git repositories and continuous deployment pipelines.

GitOps extends Infrastructure as code principles by integrating Git-based version control with automated deployment and monitoring, ensuring that the system state is always synchronized with the repository.

~~Key principles of GitOps~~

1. Declarative Configuration

Role of GitOps in Modern DevOps practices.

Commit you can revert back to

(i) Faster and Reliable Deployments: GitOps automates rollouts and rollbacks, reducing deployment failures.

(ii) Improved Collaboration and Transparency: Developers, DevOps, and SRE teams can collaborate effectively using Git-based workflows.

(iii) Enhanced Security and Auditability: Every infrastructure changes is versioned, traceable, and auditable.

(iv) Reduced configuration Drift: Automated reconciliation ensures that the actual system state matches the desired state.

(v) Infrastructure as Code Integration: works well with tools like Kubernetes, Terraform, Helm, and Ansible for infrastructure automation.

MST-1

PYQ
(2M) Enlist four significant DevOps tools and phases.

① Git (Version - Control - Continuous Development)

~~Phase~~: Git is a distributed version control system used to track changes in source code. It helps in collaboration, code management, and maintaining a history of modifications.

Example: GitHub, Gitlab

② Jenkins (Continuous Integration) - CI/CD pipeline):

~~Phase~~: Jenkins is an automation tool used to integrate code changes automatically. It allows developers to test and build their application continuously.

Example: Automates testing, builds and deployments in a DevOps pipeline.

③ Docker (Containerization - Continuous Deployment)

~~Phase~~: Docker is a containerization tool that packages applications with dependencies to run in any environment.

- It ensures consistency across different development, testing and production environment.

Example: Running microservices and deploying scalable application.

④ Kubernetes (Orchestration - Continuous Monitoring & Management)

~~Phase~~ Kubernetes is a container orchestration platform that automates the deployment, scaling and management of containerized applications. It ensures load balancing, auto-scaling, and fault tolerance.

Example: Managing Multiple Docker containers in a production environment

and DevOps phases

① planing ② Development ③ Build
④ Test ⑤ Release ⑥ Deploy ⑦ Operate
⑧ Monitor

pyq
2M Justify the utility of Git and Github.

① Git: Git is a distributed version control system that helps developers track changes in code, collaborate efficiently, and manage different versions of a project.

Utility of Git:

- Version Control: Tracks modifications/changes to the source code over time.
- Coollaboration: Multiple developers can work on the same project simultaneously.
- Branching and Merging: Enables developers to work on different features separately and merge them later.
- Backup & Recovery: Allows rollback to previous versions in case of errors.

Utility of Github:

- Remote Repository Hosting: Stores Git repositories online for easy access.

- collaboration & Code ~~Code~~ Sharing: Developers can contribute to open-source and private projects.
- pull-Requests & Code Reviews: Facilitates reviewing and merging code contributions.
- CI/CD integration: Works with tools like Jenkins, Github Actions for automated builds and deployments.
- Security & Access Control: provides features like branch protection, role-based ~~access~~ access, and issue tracking.

(AM) Illustrate various steps involved for moving the process from working to staging area and the concept of unstaging.

① working directory (untracked / Modified states): This is where you make changes to your files. Any new or modified files remain untracked or modified.

`git status`

② Staging Area (Index): The staging area containing files that are ready to be commited. You move files from the working directory to the staging area.

To stage a specific file

`git add filename`

`git add .` for multiple file.

Step 3: Commit ing
are staged, Th
repository.
To

Step 3: Committing to repository: once changes are staged, they can be committed to the git repository.

To commit staged changes:

```
git commit -m "Commit message"
```

④ Unstaging a file & moving back to Working dir)

If you mistakenly added a file to the staging area and want to remove it.

```
git reset HEAD filename
```
for a specific file

```
git reset HEAD .
```
for all files.

This moves the file back to the working dire without discarding changes.

⑤ Discarding changes (undoing working directory changes)

To discard changes in the working directory and revert to the last committed version.

```
git checkout -- filename
```
or

```
git restore filename
```

Concept of unstaging: Unstaging refers to the process of removing a file from the staging area without deleting the actual changes in the working directory.

(Pyq) Compare and contrast continuous integration, continuous delivery, and continuous deployment. How do they contribute the DevOps pipeline.

Feature	CI	CDelivery	C ~~Development~~ Deployment
Definition	Automates code integration and testing.	Ensures the code is always in a deployable state.	Automates the release process, deploying to production.
primary Goal	Detect integration issues easily.	Deliver stable, production ready code	Deploy every changes automatically.
Automation level	Automates build and Testing.	Automates testing and deployment	fully automated deployment to production.
Human intervention	Required for deployment.	Required for production deployment approval	No manual intervention.
Risk level	Low	Medium	High
impact on Devops	Improves code quality, detect errors	Speeds up the release cycle	Provides rapid feedback,
Key tools	Github, Jenkins	CircleCI, ~~Github~~ Gitlab CI/CD.	kubernetes, Docker
Rollback Mechanism	No Needed	Deployment can be delayed. (manual rollback)	Needs automated rollback strategies in case of failure

Contribution to the DevOps pipeline:

① Continuous integration: Ensures that new code changes are frequently merged, tested, and verified, reducing integration conflicts.

② Continuous Deli... build is tested ... improving ...

③ Cont... De...

② Continuous Delivery: Ensures that every successful build is tested and can be deployed at any time, improving release management.

③ Continuous Deployment: Fully automates the release process, enabling frequent and faster updates to production, ensuring rapid innovation.

9M
PYQ Explain the concept of "Immutable infrastructure" and its benefits in a devops environment. How does it differ from traditional infrastructure management?

⇒ Already done before in assignment

8M
PYQ Discuss branching? Why it is required? How to create remote branches and branches from existing one? Share two examples.

Branching: Branching in Git allows developers to work on different features, bug fixes, or experiments in isolation without affecting the main codebase/branch and can later be merged into the main branch

Why is Branching Required?

(i) Parallel Development: Different team members can work on multiple features simultaneously.

(ii) Code isolation: Changes in one branche do not affect others until merged.

(iii) Experimentation: Developers can try new ideas without disrupting the main project.

(iv) Bug fixing: Issues can be resolved in seprate branches without affecting ongoing development.

① Better collaboration: Team members can work independently and merge changes when ready.

How to create Remote Branches and Branches from existing ones?

1. Creating a new Branch from the current Branch:

```
git branch branchName
```

② Switching to the New Branch:

```
git checkout branchName
```

or

```
git switch branchname
```

③ Creating and switching to a new Branch in one Command.

```
git checkout -b new-branch
```

④ Pushing the New Branch to a Remote Repository:

```
git push -u origin branchName
```

This pushes the new-branch to the remote repository and sets up tracking.

⑤ Creating a Branch from an existing Branch.

If you want to create a new branch from an existing, first switch to develop.

```
git checkout develop
git branch branch_Name
git Checkout branch_Name
```

single Command

```
git checkout -b branch_Name develop
```

Examples:1 login page in

Examples 1: Creating a feature Branch and Pushing it to Remote.

```
git checkout -b feature-login
git add .
git commit -m "commit msg"
git push -u origin feature-login
```

This creates a feature-login branch, commits changes, and pushes them to the remote repository.

Example 2: Creating a Bug fix Branch from the develop branch to fix Bug.

```
git checkout develop
git checkout -b bugfix-issue123
git add .
git commit -m "bug fixed"
git push -u origin bugfix-issue123
```

This creates a bugfix-issue123 branch from develop, commits the fix, and pushes it to remote.

2nd MST paper

2M

Q4) Explain the function of git config command.

The git config command is used to configure Git settings at various levels, such as user information, editor preferences, and repository-specific options. It allows users to define how git behaves and interacts with repositories.

Common Uses:

1. Set User details (Global level)

```
git config --global user.name "Raufhem"
git config --global user.email "r.abed@gmail.com"
```

② check current configurations:

`git config --list`

③ Retrieve a specific configuration:

`git config user.name`

④ unset a configuration setting:

`git config --unset user.name`

PYQ How to view the commit history in Git?

ans ⇒ you can view the commit history in Git using the following commands:

1. Basic Commit History: `git log`

This shows the commit history with details like commit hash, author, data and commit message.

2. Compact History (one line per commit):

`git log --oneline` This displays each commit of a single line with a short hash and commit message.

③ Graphical Representation:

`git log --oneline --graph --all --decorate`

This shows a visual graph of branches along with commit history.

④ Commit history with file changes: `git log -p`

This shows detailed changes (diff) made in each commit.

⑤ View commit history of a specific file:

`git log --filename`

4m
Py⑨ Compare and contrast continuous integration with continuous deployment.

ans:⇒ already done before

4m
Py⑩ Explain the working directory and staging area with example.

① Working Directory: The working directory is where you make changes to your project files. It contains the actual files and folders of your project. Any modifications, additions or deletions occure here before being staged or committed.

② Staging Area(index): The staging area is an intermediate space where changes are prepared before committing. when you use git add, changes move from the working directory to the staging area, meaning Git is tracking them but hasn't yet permanently recorded them in the repository.

Example scenario.

step1: initialize a git repository

`git init` creates a new Git repo.

step2: Create and Modify files: file.txt // working directory

step3: check Git status

`git status`

step4: Add file to staging Area

`git add file.txt` // staging area

Step 5: Commit the changes

git commit -m "Added a new file"

Step 6: Modify file Again

Step 7: stage changes

`git add file.txt`

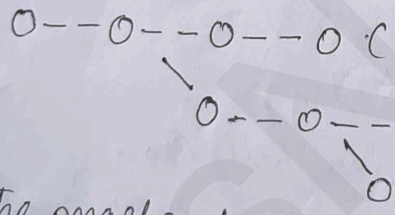
Step 8: Commit again: `git commit -m "message"`

4M

Q9) Illustrate the concept of branching strategy in git. Also explain its working with master branch and the HEAD.

Ans => A branching strategy in Git is a workflow or approach that defines how branches are created, managed, and merged in a project. It helps teams collaborate efficiently by organizing the development process and maintaining code stability.

Illustration of Branching in git



- The master branch containing the stable production-ready code.
- A feature-branch is created from master to work on new features.
- A bugfix-branch can be created from feature-branch to fix specific issues.
- Once the feature is complete, the feature-branch is merged back into master.

Working of branching strategy with master branch and the HEAD.

① The master (or main) Branch: This is the primary branch where the final version of the code resides. Typically, all branches merge into master once they are tested and stable.

(PY) HEAD: HEAD is a pointer that refers to the latest commit in the currently checked-out branch. If you switch to a different branch, HEAD moves to point to the latest commit in that branch.

Example: `git checkout master`

HEAD now points to the latest commit in master.

`git checkout feature-branch`

HEAD now points to the latest commit in feature-branch.

(PYQ) Discuss the concept of devops in detail. Explain its working. Also explain the various devops tools in detail.

Note:
- Concept of devops (Done)
- working is nothing but its (Devops lifecycle (Done))
- various devops tools (already done 4 tools enough).

Some Additional topics

Building and maintaining new branches

Building a new branch and maintaining it

Steps to create a new branch

1. check the current branches and switch to the latest branch.

```
git branch
git checkout main
git pull origin main
```

2. Create a new branch: This command creates a new branch called feature-branch.

```
git branch feature-branch
```

3. Switch to the new branch: all changes will be made in the feature branch.

```
git checkout feature-branch
```

4. Make changes and commit them

```
git add .
git commit -m "Message"
```

5. push the branch to the remote repository

```
git push origin feature-branch
```

6. Merging Back to the main: once the feature is complete and tested, you merge it back to main

- switch to main `git checkout main`
- Merge the feature branch `git merge feature-branch`
- push the main branch: `git push origin main` → also pull

merged back into master

• Delete the feature branch

```
git branch -d feature-branch
```

(*) cloning and Remotes, cloning repositories

cloning : cloning in Git means creating a local copy of a remote repository. when you clone a repository, Git downloads all the files, history, branches and commits from remote server to your local machine.

(a) command to cloning a repo (using HTTPS)

```
git clone <repository-link>
```

Example :

```
git clone https://github.com/CoderRaufhan/Dempro.git
```

(b) cloning using (SSH) : for SSH cloning, you must set up SSH keys. steps

1. Add your SSH key to GitHub.
2. Use the SSH URL.

```
git clone git@github.com:username/repository.git
```

Example :

```
git clone git@github.com:CoderRaufhan/Dempro.git
```

• Remotes : A Remote is a reference to a repository stored on a remote server. Remotes allow you to push and pull changes btw local and remote repo.

checking @ Remotes
To see the remotes linked to your repo.

```
git remote -v
```

⭐ pulling and pushing Repositories in Git

① pushing a repository:

Step ①: Initialize a Git repository

```
git init
```

Step 2: Add Remote Repository

```
git remote add origin <remote_repository_url>
```

Step 3: `git add .` stage changes

Step 4: Commit changes

```
git commit -m "message"
```

Step 5: `git push origin main`

Example: Suppose I have created a project and want to push it to github.

```
git init
git remote add origin https://github.com/coderraushan/pro.git
git add .
git commit -m "msg"
git push origin main
```

② Pulling a Repository: pulling is the process of downloading updates from a remote repo to your local machine.

Steps to pull a Repository

① Navigate to your Repo.

```
cd myproject
```

② Pull latest changes from Remote Repository

```
git pull origin main
```

Example: If I am working on a project with a team and want to get the latest changes.

```
git pull origin main
```

This command fetches and merges changes from the main branch of the remote repo into your local branch.

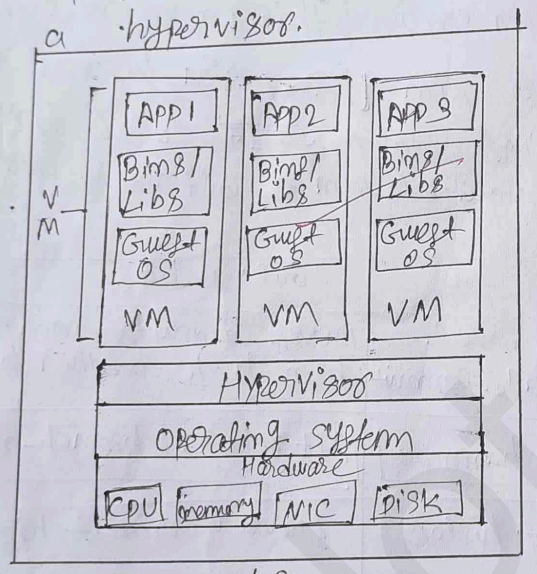
(PYQ) Discuss four major difference between Git merge and Git rebase

Feature	Git merge	Git rebase
Definition	Combines two branches and creates a merge commit	Moves commits from one branch to another branch
Command example	`git merge <branch-name>`	`git rebase <branch-name>`
Commit history	Non-linear commit history	Linear commit history
Safety	Safe for public/shared branches	Safe for local/private branches
Merge commit	Yes, it merges commit.	No, it does not merge commit

Dockers

(PYQ) Difference b/w virtualization and containerization.

Virtualization: Virtualization is the process of creating a virtual versions of computer resources such as hardware platforms, operating systems, storage devices, and network resources. It is like creating a software-based replica of a physical machine. It allows you to run multiple virtual machines on a single physical machine using a special software called a hypervisor.



How virtualization works:

(i) A physical machine (hardware) runs a piece of software called a hypervisor.

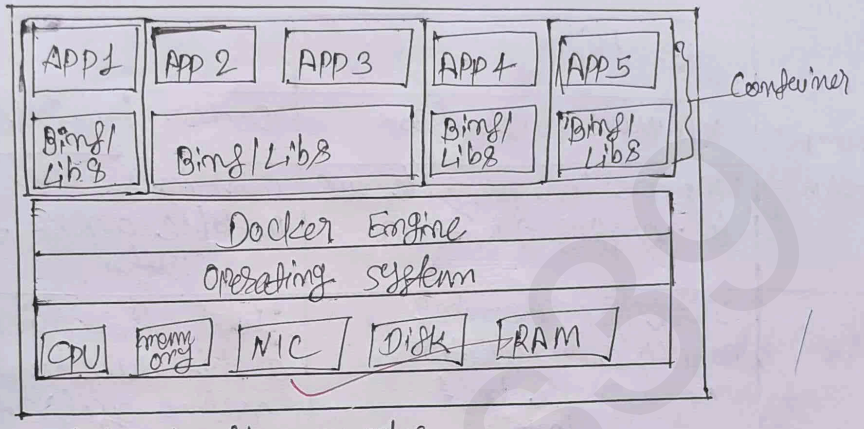
(ii) The hypervisor sits between the hardware and the operating systems.

(iii) It splits the physical resources (CPU, memory, storage) into multiple isolated virtual environment, called virtual machine.

(iv) Each VM acts like a seprate physical computer, running its own OS and applications.

* Containerization packing an applica single unit ca the same ope from each something is a

*Containerization: Containerization is the process of packing an application and its dependencies into a single unit called a container. Each container shares the same operating system kernel but is isolated from each other, providing a portable and consistent runtime environment for applications. Containerization is a lightweight form of virtualization.



How containerization works

(i) Instead of virtualizing the hardware, containers virtualize the OS.

(ii) A container runtime manages containers.

(iii) All containers shares the same OS kernel, but each × things in its own environment.

(iv) Containers are much lighter and faster than VMs.

(v) containers are designed to be scalable allowing you to quickly scale up or down.

* difference between virtu... and containe...

features	Virtualization	Containerization
operating system	Each VM has its own guest OS.	Containers shares the host OS kernel.
Size	Large (GB)	small (MB)
Startup Time	Slower (minutes)	Faster (milliseconds)
Resource usage	High	low
Example	VMware, VirtualBox	Docker, Kubernetes
portability	VMs are less portable due to varying guest OS.	Containers are highly portable across different systems
use cases	Ideal for running multiple applications on a single host	Ideal for microservices architecture and cloud-native application
performance	It has ~~higher~~ lower performance due to multiple OS overhead instance of.	It has ~~lower~~ higher performance due to ~~overhead~~ shared OS kernel

(iii) Running allows yo... Linux

(iv) I... s...

(Q6) Discuss the usage of hypervisor.

Ans=> A hypervisor is a software, hardware, or firmware component that allows multiple virtual machines to run on a single physical machine. It is also called a virtual machine monitor or VMM.

Usage of ~~Hper~~ Hypervisor:

1. Server consolidation: Instead of having one server for one application, you can run multiple applications on a single physical server.
2. Testing and Development: Developers/testers can quickly create and destroy different environments.

(iii) Running Multiple operating systems: Hypervisor allows you to run multiple operating systems (windows, Linux, macOS) together on a single computer.

(iv) Isolation and security: Each VM is isolated which does not affect others.

(v) Load Balancing and High Availability: Hypervisors work with load balancers to move VMs b/w servers. If a server crashes, VMs can be migrated automatically to another server.

(AQ1) Explain docker container in details covering their key features, benefits & use cases.

ans → Docker Containers:
A Docker container is a lightweight, standalone executable package that includes everything needed to run a piece of software - including the code, runtime, system tools, libraries and settings. Containers isolate software from its environment, ensuring it works uniformly. Containers share the host machines operating system kernel. Containers don't carry an entire OS with them making them much lighter and faster.

Key features of Docker Containers

(i) Lightweight: Since it shares only the OS kernel, so it need less resources that is why it is lightweight.

(ii) portability: It can run on any system that supports Docker.

(iii) Isolation: Each Containers runs independently, with its own filesystem, cpu, memory and process space.

(iv) scalability: Easy to scale applications horizontally using orchestration tools like Kubernetes.

(v) Version control: Docker images can be versioned and rolled back easily.

(vi) Security: provides process isolation, resource limits and reduced attack surface compared to full VMs.

(vii) Rapid Development: Containers can start almost instantly.

(viii) Resource Efficiency: Uses less CPU, memory and storage compared to traditional VMs.

Benefits of Docker Containers

(i) Isolation and Security: problem in one container doesn't affect others, containers are isolated from the host and each other.

(ii) Cost-Effective: Better resource utilization and faster development mean lower operational and infrastructure costs.

(iii) Improved CI/CD Processes: Docker integrates well with Continuous integration and Continuous Deployment (CI/CD) pipeline, speeding up the development cycle and reduce manual errors.

(iv) portability: Docker containers can run on any system that supports Docker ensuring consistency across development, testing and production environments.

(v) Microservices support: Docker fits naturally with micro-services architecture, allowing easy scaling of individual services independently.

(vi) Lightweight: Containers share the host OS kernel, meaning they are much lightweight than traditional virtual machine.

(vii) Version Control and Rollbacks: Docker images can be versioned, allowing you to roll back to previous versions easily.

Use cases of Docker Containers

1. Microservices Deployment: Containers are perfect for microservices because each services can run in a separate container with its own dependencies.

2. Simplifying Development Environment setup. Developers can get started immediately without worring about installing database, backend Servers etc. locally.

3. Continuous integration / Continuous Deployment (CI/CD): Developers freaquently merge their code changes into a shared repository, automated tests and builds are run to quickly detect problems. After code passes the CI process, it is automatically deployed to production.

4. Testing and Debugging: Quickly spin up isolated environments to test new versions of applications without affecting the production environment.

5. Multi-cloudy Deployments: Since container are portable companies can move workloads easily between on-premises and cloud environments.

PYQ+ASS Describe the distinctions between docker containers and virtual machines as well as their impact on contemporary software development methodology. include relevant example to illustrate your points. Also. how they contribute to modern software development practices Provide example where applicable.

Features	Docker container	virtual Machines
Definition	A lightweight, standalone, executable package that includes everything needed to run a piece of software	A full-blown computer running inside another computer, with its own OS, kernel and hardware.
Resource Usage	Shares the host OS kernel, consumes fewer resources and start faster.	Runs a full guest OS on virtualized hardware, more resource use and slower to boot.
Isolation Level	process-level isolation (within the same OS)	full isolation (separate OS for each VM)
Size	Lightweight	Heavyweight
Boot Time	A few seconds	Several minutes
Management	Easy to manage with tools like Docker CLI, Compose, kubernetes	Managed by hypervisors like VMware
Use cases	Microservices, scalable applications.	Running different operating systems.
Operating System	It shares the host OS kernel	It has its own guest OS over a host OS
Scalability	Highly scalable.	Less scalable.

Impact on contemporary software Development methodology.

1. Faster Development and Deployment: Docker enables developers to build, test and deploy application faster. Developers can package an app and all its dependencies into a container and ship it anywhere.

VMs, being heavy, are slower to spin up and update. Earlier, setting up development environment used to take hours or days.

Example: A
Container
Containe
envi
(2)

Example: A developer builds a Node.js app in a Docker container, tests it locally, and pushes the same container to production without worrying about environment mismatches.

② Microservices Architecture:

Docker containers fit naturally into microservices, where applications are broken into small, independent services communicating over APIs.

VMs are less efficient for microservices because spinning up a new service would mean creating a new VM, which is costly and slow.

Example: Netflix uses Docker to manage thousands of microservices that independently scale and update without downtime.

How Docker Containers contribute to Modern Software Development practices.

① Microservices Architecture:

• problem: Monolithic applications are hard to scale and maintain.

Docker solution: Applications can be split into small, independent containers (microservices) that are easier to develop, scale, and deploy individually.

Example: An e-commerce site where cart service, payment service, and Search service are all running in separate containers.

② Consistency Across Environments:

problem: applications often behaved differently across dev, test and production environment.

Docker solution: Containers bundle the application along with everything it needs (libraries, configurations, dependencies) so, it runs exactly the same everywhere.

Example: A python app tested in a container on a developer's laptop can be deployed to AWS, Azure or GCP with no changes.

③ Scaling Applications Easily

- problem: Scaling traditional apps needs new hardware or VMs - time consuming.
- Docker solution: You can scale containers horizontally in seconds using container orchestrators like kubernetes.

Example: A web service under heavy traffic can autoscale its Docker container on kubernetes without downtime.

(P&A) How do containers communicate within Docker?

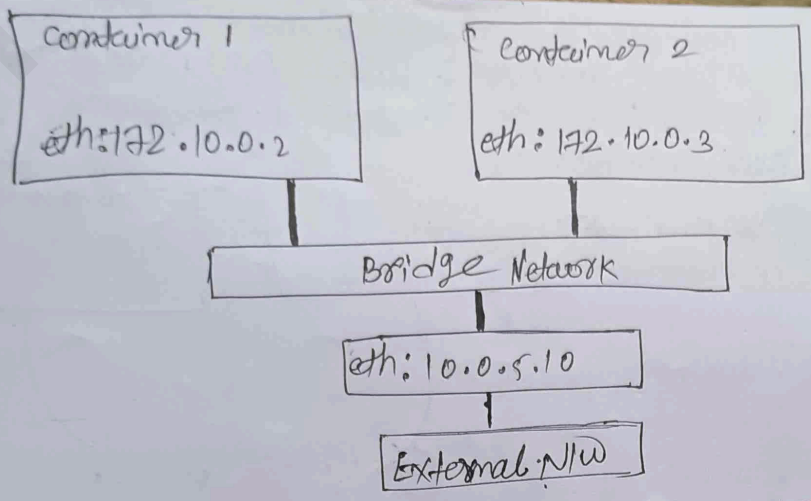
In Docker, containers communicate mainly through networking

① Bridge Network (default): By default, Docker creates a bridge network called bridge. Containers launched without specifying a network are attached to this bridge. They can communicate via IP addresses inside the bridge network. However, they don't automatically know each other's names. You usually have to link them manually

Example:

```
docker run --name container1 myimage
docker run --name container2 --link container1 myimage
```

Here --link is old and deprecated -

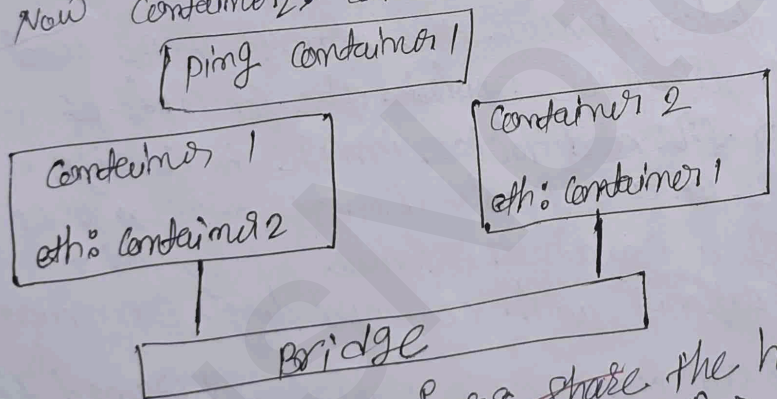


② User-Defined Bridge Networks (recommended): If you create your own bridge network, containers can discover each other by name.

Example:

```
docker network create my-network
docker run --name container1 --network my-network myimage
docker run --name container2 --network my-network myimage
```

Now container2 can reach container1 just by its name.



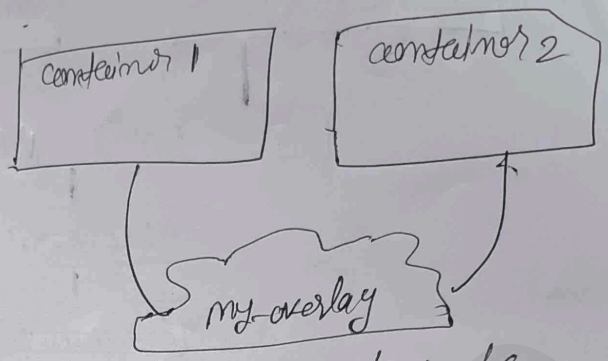
③ Host Network: Containers share the host's network stack, No isolation between container and host, useful for very high performance or when the container needs to behave exactly like a host process. Uses host IP address directly.

Example: `docker run --network host myimage`

But you lose network isolation b/w containers.

④ Overlay Network (for Swarm/Kubernetes):
It is used when containers are spread across multiple Docker hosts. Docker handles the communication using VXLAN tunneling.

~~⑤ Communication ports~~



⑤ Communication ports: If a container needs to be ~~accessed~~ accessed from outside (like from your browser), you have to expose or publish ports.

Ex ⇒ `docker run -p 8080:80 myimage`

this maps the container's port 80 to your machine's 8080.

(PYQ) what is docker, Discuss the various steps used in docker container life cycle. Explain the concepts of docker compose, docker file and docker image in details. and what steps are involved in creating, stopping, start and removing a container, provide details elaborations with corresponding commands

ans ⇒ Docker is an open-source platform designed to automate the deployment, scaling and management of application using containerization. It allows you to package an application and all its dependencies (libraries, systems, tools, code, runtime) into a single unit called a container. This container ensuring that the application runs consistently across any environment that supports Docker.

Key Compone
① Docker Daemo
the backgound pr
or server. It
Running Dock
and

Key Components of Docker

① Docker Daemon (dockerd): Docker Daemon is the background process that runs on your computer or server. It is responsible for Building Docker images, Running Docker images, Managing Docker networks, volumes, and other Docker objects also communicating with Docker clients like the Docker CLI. Whenever you type a Docker command (docker run, build etc.) it talks to the Docker Daemon to actually perform the action. Ex ⇒ `docker run hello-world`

② Docker client (docker): The Docker client is the command-line interface (CLI) that allows you to interact with the Docker Daemon. It's the tool you use to send commands to the Docker system/Daemon, which then triggers actions like running containers, building images or managing resources.

Ex ⇒ `docker run -d nginx`

③ Docker Image: Docker image is a blueprint or template that defines how a container should be created. It contains everything that needed to run an application such as Operating system, runtime environment (node.js), Code, Libraries and dependencies, configuration files. Images are built using a Dockerfile, which is a text file containing instructions on how to create the image.

Ex ⇒

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

④ Docker Containers: Already done before

⑤ Docker Registries: A Docker Registry is a centralized storage system where Docker images are stored and can be accessed by users. Docker registry allows you to upload (push) images and download (pull) images from the registry, enabling the sharing and distribution of Docker images. Ex⇒ Docker Hub (like GitHub)

Ex⇒ you can pull the nginx image from Docker Hub

```
docker pull nginx
```

you can push the image to your repo on Docker Hub

```
docker push myusername/myapp:v1
```

⑥ Docker Networks: A Docker Network allows Docker containers to communicate with each-other, the host machine or external n/w. By default, Docker creates its own networks for containers, but you can create and configure your own networks (bridge, host, overlay).

⑦ Docker Volumes: A Docker volume is a storage mechanism for persisting data generated and used by Docker Containers. Volumes are stored outside of the container's filesystem, which ensures that the data is not lost when a container is removed or recreated. They are primarily used for persisting data across container restarts, sharing data between containers, and providing a way to back-up, restore or migrate data.

⑧ ~~Docker file:~~ Dockerfile: A Dockerfile is a ~~text~~ file that contains a set of instructions used to build a Docker image. It defines how an image is created, including specifying the base image, the environment configuration, the files to be copied, the software to be installed.

and how the application should run when the container starts. Dockerfile is a blueprint for building Docker images.

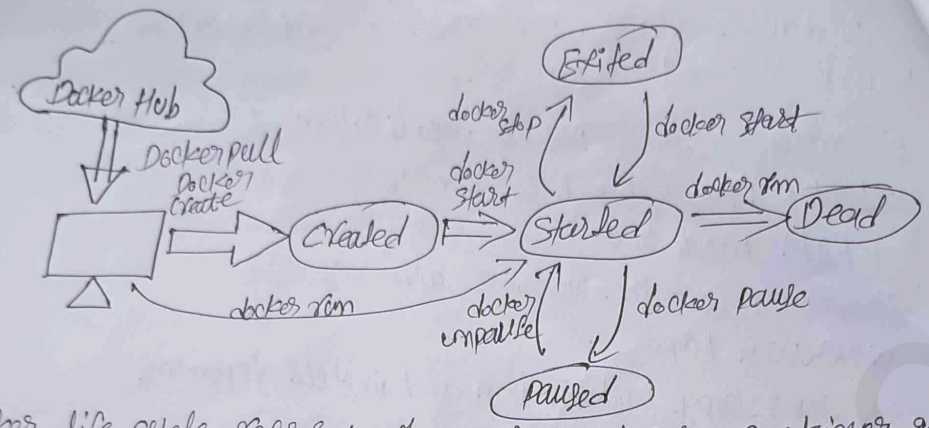
Example ⇒ for a Node.js application

```
# Step 1: Set the base image
FROM node:14
# Step2: Set the working directory
WORKDIR /app
# Step3: Copy package.json and install dependencies
COPY package.json /app
RUN npm install
# Step4: Copy the application code
COPY . /app
# Step 5: Expose the port the app will run on.
EXPOSE 3000
# Step 6: Set the command to run the app
CMD ["npm", "start"]
```

(9) Docker Compose: A tool for defining and running multi-container Docker applications. It simplifies the process of managing multiple containers by using a configuration file (docker-compose.yml), where you define the services, networks, and volumes your application needs. Docker Compose makes it easy to set-up, deploy, and manage complex applications that require multiple containers to work together.

(10) Docker Swarm (Kubernetes): For managing and scaling multiple docker containers across multiple hosts. Swarm is Docker's native orchestration solution, while kubernetes is a more comprehensive and widely adopted alternative tool.

* Docker container lifecycle



Docker life cycle refers to the various stages a container goes through.

1. Create (Created State): When you run a Docker command like `docker create <image-name>`.
Docker creates a container from the specified image, but does not start it yet.

2. Start (Running State): When you start a created or stopped container `docker start <container-id>`
The container is now running. It's main process inside starts executing. A container can also be created and started in one command. `docker run <image-name>`

3. Pause (Paused State): You can pause a running container `docker pause <container-id>` This suspends all processes inside the container.
You can later unpause it
`docker unpause <container-id>`

4. Stop (Stopped State): To stop a running container `docker stop <container-id>` This command gracefully shuts down the main process inside the container.

5. Restart (Running State) Again container. `docker`

This is equi

6. Kill (Forcefully corresponding

⑤ Restart (Running Again): you can restart a stopped container. `docker restart <container_id>`

This is equivalent to `Stop + Start`

⑥ Kill (forcefully stopping): If a container is not responding, you can kill it.

`docker kill <container_id>` This command sends a SIGKILL signal, terminating the container forcefully.

⑦ Remove (Deleted State): To completely remove a container

`docker rm <container_id>` This deletes the container and free up resources.

You cannot remove a running container directly without stoping it first, use `-f (force)` flag to do so.

`docker rm -f <container_id>`

Steps for creating, start stopping and removing a container.

① Create (Run) Docker Container: A Docker container is created and started by running the command.

`docker run` This command also pulls the image from the Docker registry and runs it in a new container.

Command: `docker run <options> <image_name>`

→ image name

→ flags that customize the behaviour of the container

② List running containers: once containers are running, you might want to view a list of currently running containers

command: `docker ps` this command shows only the containers that are running.

It will list container Ids, names, ports.

③ Stop a Running Container:- Stopping a container will gracefully halt the process running inside it. You can stop a running container by using the following command:

```
docker stop <container_name_or_id>
```

```
docker stop mynginx
```

④ Start a stopped Container:- If you have previously stopped a container and want to restart it, use the docker start command.

```
docker start <container_name or Id>
```

```
docker start myngix
```

⑤ Remove (Delete) a Docker Container:- Once you no longer need a container, you can remove it using the docker rm command. Note that you must stop the container before you can remove it

```
docker rm <container name or id>
```

```
docker rm myngix
```

⑥ Remove stopped container:- you can also remove a container immediately after stopping it with the --rm

```
docker run --rm <options> <image>
```

--rm automatically removes the container when it stops

⑦ Check logs of a Running Container:- If you need to check the logs of a container to view its o/p.

```
docker logs <container_name or id>
```

Define persi...
By defa...
container in...
one stage...
data...

(Q10) Define persistent storage, Explain its types.

By default, when you create and run a container in Docker, all the data inside the container are stored. If the container is deleted or crashes, the data is lost. persistent storage is a way to store data permanently, even if the container is stopped, removed or crashes.

In Docker, persistent storage is mainly handled using two methods

① Volumes ② Bind mounts

① Volumes: Volumes are the directories or files that exist on the host filesystem and are mounted to the containers for persisting data generated by containers. They are stored in the part of the host system managed specifically by Docker and it ~~should~~ can not be modified by non-Docker processes.

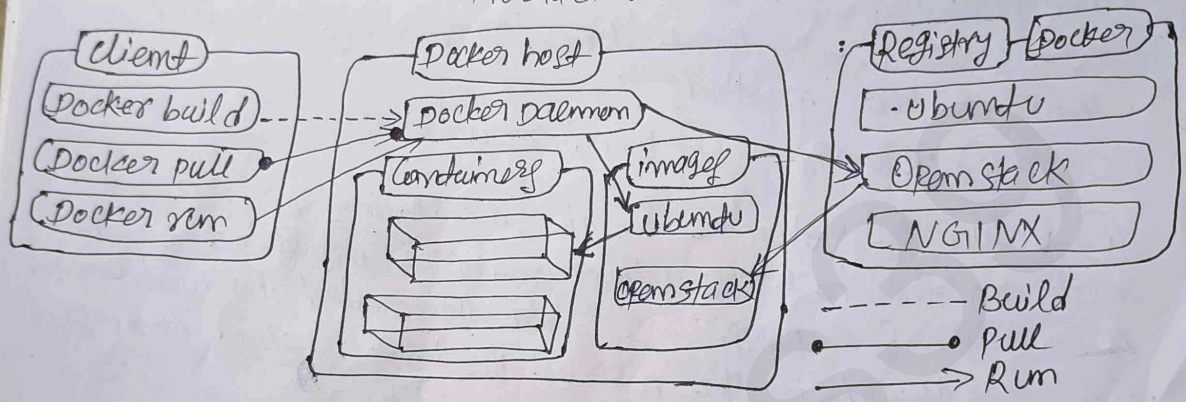
Command to create docker volume

`docker volume create <volume_name>`

② Bind mounts: This is also a storage mechanism provided by Docker to store container data on localhost, but the directory or file mounted using bind ~~mode~~ mounts can be accessed by non-docker processes as well as it relies on the host machine's filesystem having a specific directory structure available because it uses absolute path for binding.

`docker run -v /host/path:/container/path image_name`

* Architecture of Docker : Docker uses a client-server architecture. The Docker client talks with the docker deamon which helps in building, running and distributing the docker containers. The Docker client runs with the Daemon on the same system, with the helps of Rest API over a network. The docker client and daemon interact with each other.

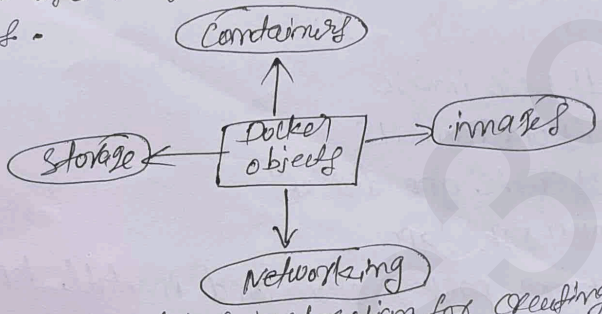


Docker client: With the help of the docker client, the docker users can interact with the docker. The docker commands uses the Docker API. The Docker client can communicate with multiple daemons. When a docker client runs any docker command on the docker terminal then the terminal sends instructions to the daemon. The docker daemon gets those instructions from the docker client inside the shape of the command and REST APIs' request. The common commands which are used by clients are docker build, docker pull and docker run.

Docker Host : A Docker host is a type of machine that is responsible for running more than one container. It comprises of the Docker daemon, images, containers, networks, and storage.

Docker Daemon: Docker daemon manages all the services by communicating with other daemons. It manages docker objects such as images, containers, networks and volumes with the help of the API requests of Docker.

Docker objects: Whenever we are using a docker, we are creating and use images, containers, volumes, networks and other objects.



Docker images: An image contains instruction for creating a docker container. It is just a read-only template. It is used to store and ship application.

Docker Containers: Containers are created from docker images as they are ready applications. With the help of Docker API or CLI, we can start, stop, delete a container. A container can access only those resources which are defined in the image.

Docker Registry: All the docker images are stored in the docker registry. There is a public registry which is known as docker hub that can be used by anyone. A Docker registry is a centralized storage system where docker images are stored and can be accessed by users. Docker registry allows you to upload (push) images and download (pull) images from the registry. Just like github.

Creating Custom Docker images, pushing images to Docker Hub.

Aim => To create a custom Docker image and push it to Docker Hub, follow these steps.

1. Write a Dockerfile : A Dockerfile contains instructions to build your image. image.

Example: A simple Nodejs app.

Dockerfile

```
# Base image
FROM node:18
# Create app directory
WORKDIR /app
# Copy package.json and install dependencies
COPY package*.json ./
RUN npm install
# Copy rest of the app
COPY . .
# Expose port and start app
EXPOSE 3000
CMD ["node", "index.js"]
```

2. Build the Docker image

Use the terminal to build the image

```
docker build -t username/node-app .
```

3. Run the Docker container (optional)

```
docker run -p 3000:3000 username/node-app
```

4. Login to Docker Hub

```
docker login
```

③ push the image to Docker Hub

`docker push username/node-app:v1`

④ pull the image

`docker pull username/node-app`

⑤ Layers of a Docker image

A Docker image is composed of multiple layers. These layers are created when commands in the Dockerfile are executed. Each layer represents a change or addition to the filesystem, such installing software, copying files or setting environment variables.

Docker image layers

① Base Layer: This is the first layer of the image and it is usually the OS. It contains the base operating system required for the containerized application.

② Intermediate Layers: These layers represent each command in the Dockerfile. Each instruction in the dockerfile (like RUN, COPY, ADD) add a new layer to image.

③ Final Layer (Top layer): This is the layer that runs when the container starts. It often contains the application or service you want to run inside the container.

Key features of Jenkins

(i) Continuous Integration and continuous Delivery: Jenkins automates the build, test and deployment processes, enabling frequent integration of code changes and faster software release.

(ii) Extensive plugin Ecosystem: Jenkins has a massive library of plugins (1800+) to integrate with virtually any tool in the CI/CD.

(iii) Open source: Being open-source and free to use promotes a large and active community that contributes to its development and provides support.

(iv) Easy Configuration: Jenkins provides a user-friendly web interface for easy setup and management of jobs.

(v) Pipeline as Code: Defining build pipelines as code in a Jenkinsfile allows for better maintainability, versioning, and collaboration.

(vi) Integration capabilities: Seamlessly integration with numerous development tools, version control systems like Git, build tools like Maven, testing frameworks like Junit, and deployment platforms.

Integration capabilities.

(i) Version Control Systems: Git, GitHub, GitLab, BitBucket.

(ii) Build Tools: Maven, NPM, Ant.

(iii) Testing frameworks: Junit, TestNG, Selenium, Postman.

(iv) Deployment Tools: Docker, kubernetes, AWS, Azure.

(v) Notification system: Email, Microsoft Teams.

Jenkins

PYQ Discuss Jenkins as a CI/CD tool covering its key features, integration capabilities & benefits for Software development process, Also explain Jenkins Architecture with the help of diagram.

Jenkins: Jenkins is a free, open source tool that helps developers to automate the process of building, testing and deploying code. It is written in Java and runs on the Java platform. if you want to install Jenkins in your system. It is primarily used for Continuous integration (CI) and Continuous Delivery (CD) in software development.

How Jenkins works as a CI/CD Tool :

(i) Continuous integration (CI): Jenkins integrates with various version control systems like Git. It can monitor repositories for changes and trigger builds automatically upon code commits. Jenkins automates the process of compiling code, building executables and packaging applications. It supports various build tools like Maven, Jenkins can execute a wide range of automated tests, including unit tests, integration tests.

(ii) Continuous Delivery : Jenkins can automate the deployment of applications to various environments. Jenkins pipelines allow defining the entire CI/CD workflow as code in a Jenkinsfile, which can be version controlled alongside the application code. Jenkins integrates with numerous deployment tools and platforms including Docker, Kubernetes.

Benefits for Software Development Teams

(i) Faster Development cycles: Automating build, test and deploy reduces manual steps.

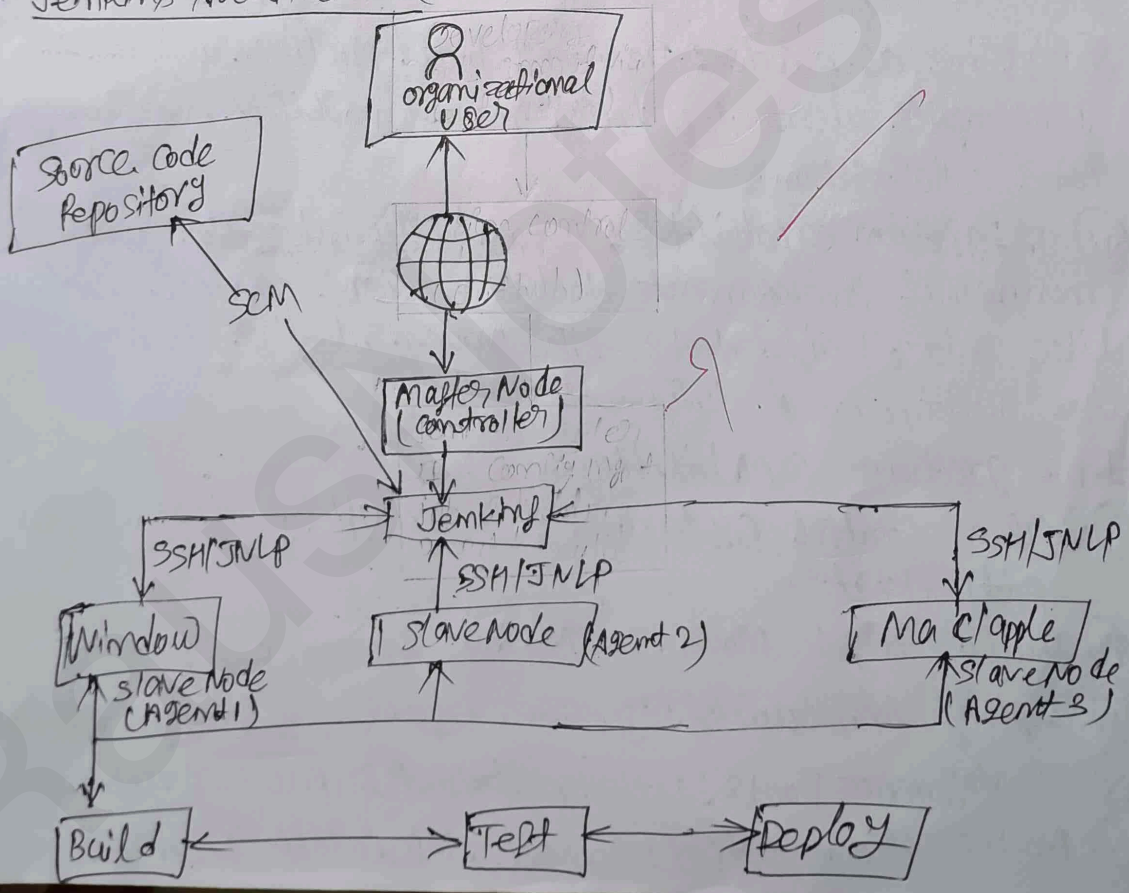
(ii) Improved Code Quality: Continuous testing and integration catch bugs early in the development process.

(iii) Increased Collaboration: Shared pipelines and dashboards improve transparency and coordination b/w developers, testers and DevOps teams.

(iv) Customization and flexibility: Jenkins can be customized to suit specific project needs.

(v) Cost-Effective: Being open-source, Jenkins has no licensing cost and is supported by a large community.

* Jenkins Architecture (Diagram):



1. Developer: Developers write code and push it to a Version control system like Git, Github. This triggers a Jenkins job via webhook.

2. Version control system (VCS): Jenkins integrates with VCS tools like Git, Bitbucket. On code changes, Jenkins fetches the updated code to build and test.

③ Jenkins Master: Jenkins Master is the central control unit of the entire jenkins setup. It is responsible for

(i) Job scheduling: It decides when and where to run jobs based on triggers.

(ii) Managing Nodes (Slaves/Agents): It connects and manages the worker Nodes (Slaves or agents) that execute the actual build and test tasks.

(iii) Storing configuration: It stores all the configuration information, including job definitions.

(iv) Presenting UI: It provides the Web-based user interface for interact with Jenkins

(v) Dispatches Tasks: It distributes build and test execution tasks to the available slave.

(vi) Monitoring slaves: It monitoring the health and availability of the connected slave nodes.

④ Jenkins Slave (Agent): Slaves are the worker machines that perform the actual work of building, testing and deploying software sent by the master. Slaves are connected to the master to receive tasks and report their status and result. This communication typically happens via protocols like JNLP (Java network launch protocol) or SSH.

⑤ Artifact Repositories: These repositories are used to store the build products (JAR files, WAR files, Docker images) produced by jenkins

1 (Pg 8) Demonstrate the utilization of jenkins for project building and integration with git repository. [Describe the process of linking Jenkins to a Git repo]

ans⇒ Utilizing Jenkins for project building and integration with a Git repository.

(1) prerequisites:

(a) Jenkins installed and running (usually at http://localhost:8080)

(b) Git installed on the jenkins server

(c) Github with a sample project (e.g. Node.js).

(d) JENKINS plugins installed.

(i) Git plugin
(ii) pipline (optional)
(iii) Maven integration plugin
(iv) NodeJS plugin (if working with Node.js)

(2) Create a New Jenkins Job:

Step1: Login to Jenkins Dashboard

Step2: Create a New item
- click on "New item"
- Enter a project ~~Name~~ (myfirstpro)
- Select Freestyle project
- click OK.

(3) Configure Git Repository:

Step1: Source ~~Code~~ Code Management
- Enter the Git repository URL (e.g., https://github.com/CoderRaghem/raufinsta.git)
- If it is a private repo, add credentials.

Step2: Branch to Build
- Specify the branch name eg (*/main)

④ Add Build steps:
For a Node.js project
- Add build step: Execute shell
- Command

```
npm install
npm run build
```

⑤ Save and Build
- click Save
- click Build Now
- observe console output for build progress.

⑥ monitor the Build
- once the build starts, a build history will appear on the project page.
- click console output to view the real-time logs of the build process. This is where you can see if the build was successful or if any errors occurred.

Q⑥ provide Example of how jenkins can be configured & utilized in real-world scenarios.

ans ⇒ Scenario: web Application Deployment (Node.js + React)

Goal: Automate build, test, and deployment processes using Jenkins whenever code is pushed to GitHub.

1. Jenkins configuration

Step1: Install Required plugin
- Git plugin
- NodeJS plugin
- pipeline plugin
- Docker

Step 2: Create a New Jenkins Job
- Select "pipeline" Job

Name it : NodeApp

Step 3: Connect to GitHub (integrate git repository)

- Repository URL : https://github.com/CoderRaughan/raughinta.git
- Credentials : Add Github if private repo.
- Branch : main or develop

(2) Jenkins Pipeline script (Jenkinsfile)

```
pipeline {
    agent any
    environment {
     NODE_ENV = 'Production'
     PATH = '/usr/local/bin:path' // path
    }
    tools {
     nodejs 'NodeJS 18'
    }
    stages {
    stage('checkout'){
      steps {
         git branch: 'main',
    url: 'https://github.com/CoderRaughan/raughinta.git'
      }
    }
    stage('Install Dependencies')
    { steps {
         sh 'npm install'
    }  }
    stage('Run Tests')
    { steps {
      sh 'npm run test'
    } }
```

```
stage('Build project'){
  steps{
    sh 'npm run build'
  }
}
stage('Deploy')
steps{ sh '''
    scp -r build /user@yourserver.com:/var/www/html/
  '''
}
}}}
post{
  success{
    echo 'Deployment successful!'
  }failure{
    echo 'pipeline failed'
  }
}}
```

(Q5) Explain Jenkins pipeline. Illustrate the steps used to demonstrate how Jenkins is used to build projects. and Elaborate on the concept of a multibranch.

ans: A Jenkins pipeline is a set of steps that jenkins uses to automate the process of moving code through stages like:

- Building
- Testing
- Delivering
- Deploying

Each stage in the pipeline represents a phase in the DevOps lifecycle

Types of Jenkins pipelines

(a) Declarative Pipeline (Recommanded)
(b) Scripted Pipeline

(a) Declarative Pipeline: Simpler and more structured use pipeline block, Easy to read and maintain.

```
Pipeline{
  agent any
  stages{
    stage('Build')
    {steps{
      echo 'Building...'
    }}
    stage('Test'){
    steps{
      echo 'Testing...'
    }}
    stage('Deploy'){
      steps{
        echo 'Deploying...'
      }}}
}
```

(b) Scripted pipeline: It use Groovy scripting, offers more control and flexibility. More complex to write and manage.

```
node{
    stage('Build'){
      echo 'Building'
    }
    stage('Test')
  {echo 'Testing'
  }
  stage('Deploy')
  echo 'Deploying...'
  }
}
```

Jenkinsfile: This is a text file that stores the pipeline code usually stored at the root of the repository.

Key components of Jenkins pipeline

agent: Defines where the pipeline will run

Stages: Blocks that contain multiple stage.

Stage: Represents a single job.

Steps: Define tasks to be executed within a stage

post: Actions that run after stages

environment: Sets environment variables for the pipeline.

parameters: Allow passing parameters to the pipeline.

Multibranch pipeline: A multibranch pipeline is a Jenkins feature that automates the creation and management of pipelines for multiple branches within a single Git repository. Instead of manually creating a seprate pipeline job for each branch, Jenkins automatically discovers branches containing a Jenkinsfile and creates a corresponding pipeline for each one.

purpose and Benefits:

- Automates the creation of pipelines for all branches
- Supports multiple branches and pull requests independently
- Automatic cleanup when a branch is deleted from the Repository

How it works:

(i) Jenkins scans the repository for branches.

(ii) for each branch containing a Jenkinsfile,

(iii) if a new branch is pushed, Jenkins automatically creates a pipeline for it.

(iv) If a branch is deleted, its pipeline job is also removed.

Illustrate the steps used to demonstrate how Jenkins is used to build projects. ⇒ already done in previous questions.

* How to revert a commit

`git revert <commit-hash>`

* delete a remote branch

`git push origin --delete feature/login`

* Detail the working with Bourne Again Shell (BASH)

ans → Working of BASH

Bash is a command-line interpreter that lets you interact with operating system by typing commands. It is commonly used in Linux and unix systems.

Bash commands

pwd: print work directory
ls : List directory
cd : change directory
mkdir: create a new directory
rm : remove files
touch: create a new file
echo : display a line of text
cat : view contents of a file

* Detail the working with Bourne Again Shell (BASH)

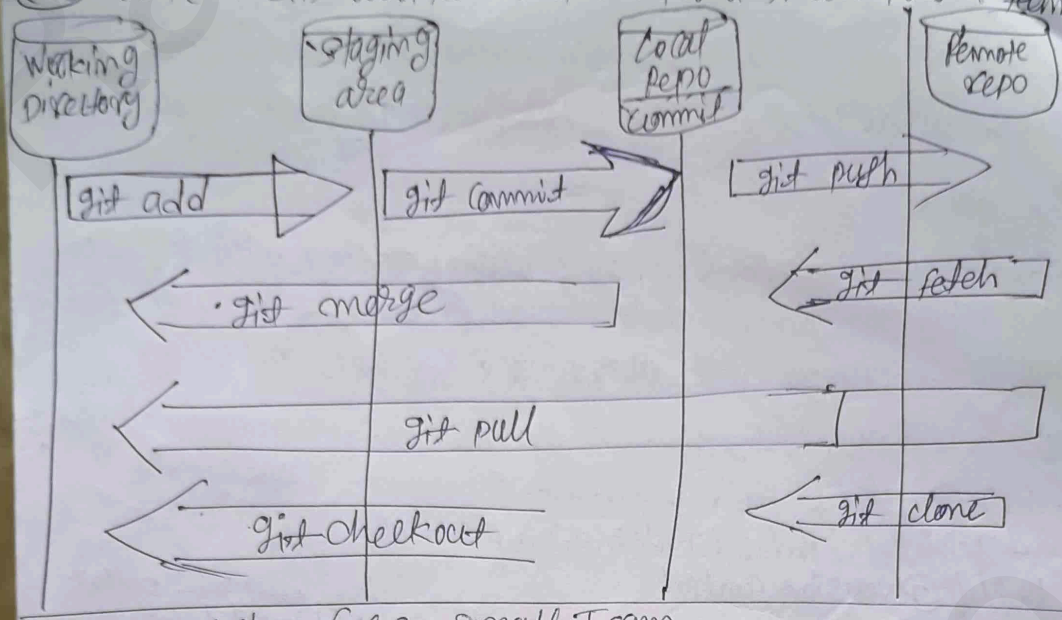
ans → Working of BASH

Bash is a command-line interpreter that lets you interact with operating system by typing commands. It is commonly used in Linux and unix systems.

Bash commands

pwd: print work directory
ls : List directory
cd : change directory
mkdir: create a new directory
rm : remove files
touch: create a new file
echo: display a line of text
cat : view contents of a file

Q15) Create a Git workflow suitable for a small project team.



Git workflow for a small Team

① main Branches:

- main: The production ready code Always deployable
- develop: The integration branch for feature before they are merged to main branch.

② Supporting Branches:

- feature/loginpage : for a new feature.
- feature/Bugfix : for minor Bug fixes.
- feature/Release : for preparing release.

③ working steps

Step1: Clone the repo : git clone <repo_url>
cd <repo_name>

Step2: Create a Integration Branch
git checkout -b develop

Step3: create a feature branch
git checkout -b feature/loginpage

Step4: work and commit locally
Add/Edit page
git add . git commit -m "Added login page"

Step5: push to remote and open a pull request
git push origin feature/loginpage

• Open a pull request into develop
• one teammate reviews and approves

Step6: Merge into develop
Use swagth and merge
Delete the feature branch.

```
git branch -d feature/loginpage
delete local branch
git push origin --delete feature/loginpage
delete remote branch
```

Step7: Merge into main
git checkout main
~~git push origin main~~
git merge
git push origin main

Step8: Merge back into develop
git checkout develop
git merge main

* Jenkins Installation and Configuration:

• Go to the oracle official web. → window installer

Step1: Install Java (JDK):
• Download JDK from oracle webs
• Install JDK
• Add new environment variable
• Verify if installed `Java -version`

Step2: Download and Install Jenkins
• Go to the Jenkins website
• Download Jenkins window installer
• Install Jenkins
• Jenkins verify: runs on port 8080

Step3: Unlock Jenkins:
• Open browser type: http://localhost:8080.
• It will ask for admin password.
• fill admin password and continue.

Step4: Install suggested plug-in and setup Admin user.
• Install suggested plugins
• Create a new Admin user (username, email, password).
• Jenkins is now ready to use.