DevOps Final PYQ

2Marks Questions

PYQ. how does Continuous Delivery impact the traditional release management process?

Ans=>

🔄 **1. Automation Replaces Manual Processes**

- **Traditional:** Releases often involve manual steps, checklists, and handoffs between teams.
- **CD:** Automates building, testing, and deployment pipelines, reducing human errors and making releases more predictable.

🚀 **2. Faster and More Frequent Releases**

- **Traditional:** Releases happen in fixed cycles (e.g., monthly, quarterly), often leading to large and risky deployments.
- **CD:** Enables on-demand, smaller, and more frequent releases, making deployments less risky and easier to troubleshoot.

✅ **3. Built-in Quality Checks**

- **Traditional:** QA/testing occurs late in the release cycle.
- **CD:** Testing is integrated into the pipeline (unit, integration, acceptance tests), ensuring each change meets quality standards before deployment.

👥 **4. Collaboration Across Teams**

- **Traditional:** Development and operations work in silos.
- **CD:** Promotes DevOps culture, where dev, ops, and QA collaborate closely to deliver software continuously and reliably.

🔐 **5. Controlled and Auditable Releases**

- **Traditional:** Release documentation and approvals are manually tracked.
- **CD:** Release pipelines can enforce automated approvals, logs, and rollback strategies, ensuring compliance and traceability.

📉 **6. Reduced Release Risk**

- **Traditional:** Big-bang releases often lead to downtime and bugs.
- **CD:** Smaller, incremental changes make issues easier to detect and fix, improving overall system stability

PYQ. How does Git optimize performance when working with large repositories?

Ans=>

⚙️ 1. Local Operations

- Git is a **distributed VCS**, so most operations (e.g., commit, diff, log) are performed **locally**, without requiring a server.
- This greatly reduces latency and makes commands fast—even on large repos.

🧠 2. Efficient Data Storage (Snapshots, not Deltas)

- Git stores **snapshots of the entire file system**, not just differences (deltas), using **hash-based deduplication** (SHA-1 or SHA-256).
- Unchanged files are referenced by their hash, saving space and improving access speed.

🗜️ 3. Compression with Packfiles

- Git uses **packfiles** to compress and store multiple objects (commits, blobs, trees) efficiently.
- Packfiles reduce disk usage and improve clone/pull/push performance.

🌿 4. Sparse Checkout and Partial Clone

- **Sparse Checkout**: Allows you to check out only specific directories of a large repo.
- **Partial Clone**: Downloads only part of the history or required data, reducing bandwidth and disk usage.

🧩 5. Incremental Garbage Collection

- Git runs garbage collection (git gc) to **clean up unnecessary files** and optimize repository storage.
- Keeps repositories compact and fast.

📇 6. Indexing and Caching

- Git maintains an **index** (staging area) and uses **caches** to accelerate operations like status checks and diffs.

PYQ. Detail the tracked and untracked files in git?

Ans=>

✅ Tracked Files

Tracked files are files that Git knows about — they are being tracked for changes. These are files that were once added to the Git repository using git add and then committed.

Tracked files can exist in three states:

1. **Unmodified**
    - These files have not changed since the last commit.
    - Git doesn't need to do anything with them unless they're modified again.
2. **Modified**
    - These files have been changed after the last commit.
    - Git notices the change but doesn't include them in the next commit until you stage them using git add.
3. **Staged**
    - These are modified files that have been staged (added to the staging area) and are ready to be committed.

❌ Untracked Files

Untracked files are files in your working directory that are **not being tracked by Git**. They have not been staged or committed before.

- These are typically new files you've created but haven't told Git to track yet.
- Git will ignore them until you explicitly add them.

@2nd paper
PYQ How does Continuous Integration contribute to enhancing the quality of software?
Explain the purpose of containerization in DevOps.
Differentiate between pulling and pushing in GIT.

✅ 1. How does Continuous Integration (CI) contribute to enhancing the quality of software?

Continuous Integration (CI) is a DevOps practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run.

- **Early Bug Detection**: Since code is integrated and tested frequently, bugs are caught early, making them easier and cheaper to fix.
- **Faster Feedback**: Developers get immediate feedback if their changes break the build or fail a test.
- **Improved Collaboration**: With frequent integrations, teams can work collaboratively without integration conflicts.

- **Automated Testing**: CI enforces testing through automation, ensuring consistent quality and reducing manual errors.
- **Reduced Integration Problems**: Smaller, incremental changes are easier to integrate than big ones done occasionally.
- **Stable Builds**: Since every change is tested, the codebase remains more stable and reliable.

✅ 2. Explain the purpose of containerization in DevOps.

Containerization is the process of packaging an application and its dependencies together in a container so it can run reliably across different computing environments.

🎯 **Purpose and Benefits:**

- **Portability**: Containers run the same way in development, testing, and production.
- **Consistency**: Ensures consistency across multiple environments by bundling everything needed to run the application.
- **Resource Efficiency**: Containers are lightweight and use fewer resources than virtual machines.
- **Isolation**: Each container runs independently, so problems in one do not affect others.
- **Faster Deployment**: Containers can start quickly, which speeds up deployment and scaling.
- **Microservices Architecture**: Ideal for breaking down apps into smaller, independently deployable services.

3.Differentiate between Pulling and Pushing in Git

Aspect	Pulling in Git	Pushing in Git
Definition	Retrieves changes from the remote repository to local repo	Sends local commits to the remote repository
Command Used	git pull	git push
Direction of Data Flow	Remote ➜ Local	Local ➜ Remote
Use Case	To update your local branch with changes from others	To share your changes with others via the remote repository
Combination Of	Combines git fetch + git merge	Only uploads commits, doesn't merge or fetch
When Used	Before starting work to ensure updated codebase	After committing local changes to sync with the remote repo

@3rd paper

PYQ. Describe the difference between git fetch and git pull.

Ans=>

Aspect	git fetch	git pull
Definition	Downloads changes from the remote repository but does not modify the working directory or the current branch.	Fetches changes from the remote and merges them into the current branch.
Command Effect	Updates remote-tracking branches (e.g., origin/main) in your local repository.	Fetches the changes and then automatically merges them into the current branch.
Merge Action	Does not merge any changes into your working directory.	Automatically merges the fetched changes into your working directory.
Use Case	Used to inspect the changes in the remote repository without affecting your local branch.	Used when you want to update your local branch and automatically merge remote changes.
Safe vs Risky	Safer, as it doesn't change your local working directory.	Riskier, since it automatically merges changes which could result in conflicts.
Workflow Integration	Often used before starting work to check if changes are available on the remote repository.	Often used when you want to pull and integrate changes from the remote repository into your current branch.

b) Describe the primary function of Jenkins in the DevOps toolchain.

🛠️ **Key Functions of Jenkins in the DevOps Toolchain**

1. **Automates Build Process:**
    - Jenkins automates the process of building the application. It compiles code, packages artifacts, and runs automated unit tests, ensuring that the build is reliable and repeatable.
2. **Continuous Integration (CI):**
    - Jenkins provides a platform for Continuous Integration by automatically integrating code from multiple developers into a shared repository. Each integration is verified by an automated build and test, ensuring early bug detection and preventing integration issues.
3. **Continuous Delivery (CD):**
    - Jenkins can extend its functionality to automate the delivery process, enabling Continuous Delivery (CD). This means Jenkins can automatically deploy built applications to various environments like testing, staging, or production once they pass the automated tests.
4. **Pipeline Automation:**

- Jenkins allows you to define and automate complex workflows using "pipelines," which represent the entire process of building, testing, and deploying an application. These pipelines are defined in code (Jenkinsfile) and can be version-controlled with the application code.

5. **Monitoring and Reporting:**
    - Jenkins provides monitoring and reporting features, such as displaying the status of recent builds, logs, and test results. It can notify developers in case of build failures or other issues, helping teams respond quickly.
6. **Integrates with Other Tools:**
    - Jenkins integrates with a wide variety of tools in the DevOps ecosystem, such as version control systems (Git), containerization platforms (Docker), deployment tools (Kubernetes), and test frameworks (JUnit).

a) How can you stop a running Docker container? Why should you remove unused Docker containers?

Ans=>

How to Stop a Running Docker Container?

You can stop a running Docker container using the docker stop command. Here's the syntax:

docker stop <container_id or container_name>

- **Example:**
    1. First, list all running containers to find the container ID or name:

        docker ps

    2. Then stop the desired container by its ID or name:

        docker stop <container_id or container_name>

Alternatively, if you want to forcefully stop a container (in case it doesn't stop gracefully), you can use the docker kill command:

docker kill <container_id or container_name>

🧹 **Why Should You Remove Unused Docker Containers?**

Removing unused or stopped Docker containers is important for several reasons:

Reason	Explanation
✅ **Free Up Disk Space**	Stopped containers still occupy disk space, and removing them can recover space for new containers or images.
🚀 **Improve Performance**	Too many unused containers can slow down Docker's operations and lead to performance degradation.

Reason	Explanation
🧹 **Maintain a Clean Environment**	Leaving unused containers can clutter your Docker environment, making it harder to manage.
⚠️ **Avoid Resource Conflicts**	Unused containers may still bind to resources like ports, networks, or volumes, causing conflicts when new containers are started.
🔒 **Security**	Unused containers could have outdated dependencies or vulnerabilities that may pose a security risk.
📦 **Efficient CI/CD Pipelines**	Keeping only the necessary containers ensures smoother, faster CI/CD pipelines by removing unnecessary overhead.

In Git how do you revert a commit that has already been pushed and made public?

Ans=> 🎯 **Steps to Revert a Public Commit in Git**

🔁 **Option 1: Revert a Single Commit**

1. **Find the commit hash** you want to revert:

   git log

2. **Revert the commit:**

   git revert <commit_hash>

3. This creates a **new commit** that undoes the changes introduced by the original one.

4. **Push the revert commit to the remote repository:**

   git push origin <branch_name>

@5th paper

c) What is the difference between staged and unstaged resources?

Ans=>

Aspect	Staged Resources	Unstaged Resources
Definition	Files that have been marked to be included in the next commit	Files that have been modified but not yet marked for commit
Command to View	git diff --cached or git status	git diff or git status
Command to Add	git add <filename>	Changes are made directly to the file but not yet staged

Aspect	Staged Resources	Unstaged Resources
Included in Commit	✅ Yes, will be committed	❌ No, won't be committed unless staged
Storage Area	Stored in the **staging area (index)**	Still in the **working directory**
Purpose	Prepares a snapshot of changes for the next commit	Lets developers continue editing without committing right away

@6th paper

Name three important DevOps KPIs.
Ans=>**1. Deployment Frequency**

- **Definition**: How often new code is deployed to production.
- **Why it matters**: High frequency indicates faster innovation and delivery.
- **Goal**: Deploy small, incremental updates often rather than infrequent large releases.

---

2. Lead Time for Changes

- **Definition**: Time taken from code commit to deployment in production.
- **Why it matters**: Shorter lead time means faster delivery of features and fixes.
- **Goal**: Reduce delays between development and delivery.

---

3. Mean Time to Recovery (MTTR)

- **Definition**: Average time taken to recover from a failure in production.
- **Why it matters**: Lower MTTR indicates higher system reliability and quicker response to issues.
- **Goal**: Quickly detect, respond to, and fix incidents.

Illustrate the benefits of version control system?
Ans=> ✅ **Benefits of Version Control System (VCS)**

A **Version Control System (VCS)** is a tool that helps track changes to files, typically source code, over time. It allows multiple developers to work together, keeps a history of every modification, and facilitates collaboration and code quality.

Here are the key benefits of using a VCS:

📌 **1. Collaboration**

- Multiple developers can work on the same project simultaneously.

- Merges changes from different team members efficiently.
- Resolves conflicts when two people change the same part of a file.

📌 2. Track Changes Over Time

- Maintains a **history of every change**, including what was changed, when, and by whom.
- Helps understand how and why the software has evolved.

📌 3. Backup and Restore

- If something goes wrong, you can **revert** to a previous stable version.
- Acts as a backup in case of file corruption or accidental deletion.

📌 4. Branching and Merging

- Create **branches** for new features, bug fixes, or experiments without affecting the main code.
- **Merge** branches back into the main project after review and testing.

📌 5. Code Review and Quality

- Tools like **GitHub** allow team members to review code before merging.
- Encourages better coding practices and team learning.

📌 6. Audit and Accountability

- Every change is logged with the developer's name and a timestamp.
- Useful for debugging, compliance, and accountability.

📌 7. Continuous Integration (CI) Support

- Works seamlessly with CI/CD tools like Jenkins, GitHub Actions, etc.
- Automates building, testing, and deployment of code changes.

DevOps Final PYQ

4Marks Questions

PYQ. Develop a step-by-step guide for creating and configuring a new Git repository for a collaborative project. Discuss considerations for repository settings and initial commit practices

Ans=> ✅ **Step-by-Step Guide: Setting Up a Collaborative Git Repository**

🔹 **Step 1: Create a New Repository (Using GitHub)**

1. Go to https://github.com
2. Click on **"New repository"**
3. Fill in:
    - **Repository name** (e.g., collab-project)
    - **Description** (optional but helpful)
    - **Visibility**: Choose Public or Private depending on team needs
    - Check:
        - ✅ *Initialize this repository with a README* (optional but useful)
        - ✅ *Add .gitignore* (select language/environment, e.g., Node, Python)
        - ✅ *Choose a license* (e.g., MIT or Apache 2.0 for open-source)
4. Click **Create repository**

🔹 **Step 2: Clone the Repository Locally**

From your terminal or Git Bash:

git clone https://github.com/username/collab-project.git

cd collab-project

🔹 **Step 3: Configure Git (First-Time Setup)**

If not already configured:

git config --global user.name "Your Name"

git config --global user.email "you@example.com"

🔹 **Step 4: Create/Organize Project Files**

- Add folders like:
    - /src – Source code
    - /docs – Documentation
    - /tests – Unit tests
- Create essential files:
    - README.md – Project overview

- .gitignore – Files/folders to exclude (e.g., node_modules/, .env)
- LICENSE – Open source license (if applicable)

🔹 **Step 5: Stage and Make the Initial Commit**

git add .

git commit -m "Initial commit: project structure and docs"

git push origin main

🔹 **Step 6: Add Collaborators**

On GitHub:

- Go to your repository → **Settings** → **Collaborators**
- Invite team members by GitHub username or email
- Set roles (Admin, Write, Read)

Explain the steps to create first Docker container using the hello- world image.

✅ **Steps to Create Your First Docker Container Using hello-world Image**

🔹 **Step 1: Install Docker**

Before running any containers, you must have Docker installed:

- Download from: https://www.docker.com/products/docker-desktop/
- Install it based on your OS (Windows, macOS, or Linux).
- After installation, verify Docker is installed:

docker --version

🔹 **Step 2: Open Terminal or Command Prompt**

Use a terminal (Linux/macOS) or Command Prompt / PowerShell (Windows) to run Docker commands.

🔹 **Step 3: Run the Hello-World Docker Container**

Now, run the following command:

docker run hello-world

🔹 **Step 4: What Happens When You Run It**

When you run docker run hello-world, Docker does the following:

1. **Checks for the image** locally.
2. If not found, it **downloads** the hello-world image from Docker Hub.
3. Creates a **new container** from that image.

Q5. Describe the process of linking Jenkins to a Git repository.

✅ **Steps to Link Jenkins to a Git Repository**

🔹 1. Install Git Plugin in Jenkins

Most Jenkins versions come with Git support pre-installed, but to confirm:

1. Go to **Jenkins Dashboard → Manage Jenkins → Manage Plugins**
2. In the **Available** tab, search for **"Git plugin"**
3. Install it and restart Jenkins if required

🔹 2. Set Git Path (if needed)

1. Go to **Manage Jenkins → Global Tool Configuration**
2. Under **Git**, set the path to your Git executable (e.g., /usr/bin/git or C:\Program Files\Git\bin\git.exe)
3. Save the configuration

🔹 3. Create a New Jenkins Job (Freestyle Project)

1. Go to **Jenkins Dashboard**
2. Click **"New Item"**
3. Enter a project name and select **"Freestyle project"**
4. Click **OK**

🔹 4. Link to Git Repository

In the job configuration:

1. Scroll to **Source Code Management**
2. Select **Git**
3. Enter your **repository URL** (e.g., https://github.com/username/repo.git)
4. If your repo is private:
    - Click **Add** under Credentials
    - Add your GitHub username/password or SSH key

🔹 5. (Optional) Specify Branch

Under "Branches to build", enter the branch you want Jenkins to monitor:

*/main # for main branch

*/develop # for develop branch

🔹 **6. Set Build Triggers (Optional but Recommended)**

You can make Jenkins automatically pull changes from Git:

- Check **"Poll SCM"**
    - Enter schedule (e.g., H/5 * * * * for every 5 minutes)
- Or use **webhooks** (in GitHub → Settings → Webhooks)

🔹 **7. Add Build Steps**

Scroll to **Build** section and add steps like:

- **Execute shell** (Linux/macOS)
- **Execute Windows batch command**
- **Invoke Gradle/Maven/npm** as per project type

Example for Node.js:

npm install

npm test

🔹 **8. Save and Build**

- Click **Save**
- Then click **Build Now** to test it
- Check **Console Output** for logs

How do all DevOps tools work together?

✅ **How DevOps Tools Work Together — Step-by-Step:**

🔹 **1. Planning & Collaboration**

- **Tools**: Jira, Trello, Confluence
- **Purpose**: Help teams define requirements, plan features, and track progress.
- **Integration**: These tools connect with repositories (e.g., GitHub) and CI tools to link code to tasks.

🔹 **2. Source Code Management**

- **Tools**: Git, GitHub, GitLab, Bitbucket
- **Purpose**: Version control and collaboration on code.
- **Integration**: Connected with Jenkins or GitLab CI to trigger builds when code is pushed.

🔹 **3. Build Automation**

- **Tools**: Jenkins, Maven, Gradle
- **Purpose**: Automate compilation and packaging of the code.
- **Integration**: Jenkins pulls code from Git and uses Maven to build it automatically.

🔹 **4. Continuous Testing**

- **Tools**: Selenium, JUnit, TestNG
- **Purpose**: Automate test execution to ensure quality.
- **Integration**: Jenkins triggers tests post-build to ensure no bugs move to production.

🔹 **5. Continuous Integration / Delivery (CI/CD)**

- **Tools**: Jenkins, GitLab CI/CD, Bamboo
- **Purpose**: Automate the process from build to testing to deployment.
- **Integration**: Detects code changes and executes the full CI/CD pipeline.

🔹 **6. Containerization & Deployment**

- **Tools**: Docker, Kubernetes, Ansible
- **Purpose**: Package apps into containers and deploy them across environments.
- **Integration**: Jenkins can push Docker images to a container registry and trigger Kubernetes for deployment.

🔹 **7. Monitoring & Feedback**

- **Tools**: Prometheus, Grafana, ELK Stack, Nagios
- **Purpose**: Monitor application health, performance, and log data.
- **Integration**: Sends real-time alerts and feedback to developers for fixes or improvements.

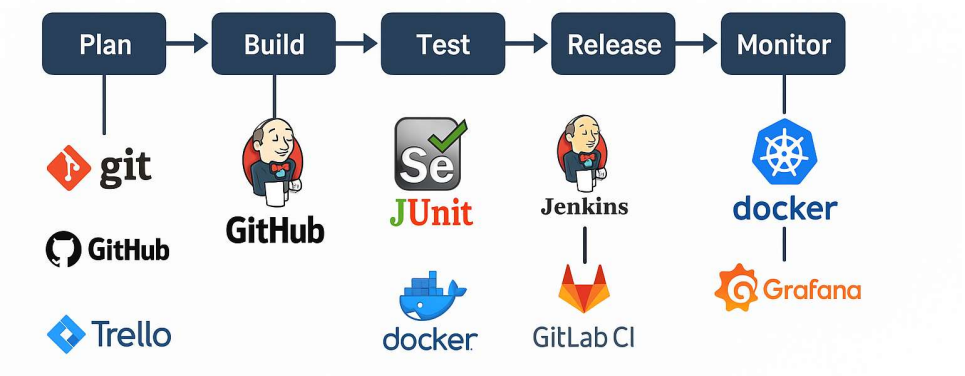
How do all DevOps tools work together?

✅ **How DevOps Tools Work Together — Step-by-Step:**

🔹 **1. Planning & Collaboration**

- **Tools**: Jira, Trello, Confluence
- **Purpose**: Help teams define requirements, plan features, and track progress.
- **Integration**: These tools connect with repositories (e.g., GitHub) and CI tools to link code to tasks.

- **2. Source Code Management**
  - **Tools**: Git, GitHub, GitLab, Bitbucket
  - **Purpose**: Version control and collaboration on code.
  - **Integration**: Connected with Jenkins or GitLab CI to trigger builds when code is pushed.
- **3. Build Automation**
  - **Tools**: Jenkins, Maven, Gradle
  - **Purpose**: Automate compilation and packaging of the code.
  - **Integration**: Jenkins pulls code from Git and uses Maven to build it automatically.
- **4. Continuous Testing**
  - **Tools**: Selenium, JUnit, TestNG
  - **Purpose**: Automate test execution to ensure quality.
  - **Integration**: Jenkins triggers tests post-build to ensure no bugs move to production.
- **5. Continuous Integration / Delivery (CI/CD)**
  - **Tools**: Jenkins, GitLab CI/CD, Bamboo
  - **Purpose**: Automate the process from build to testing to deployment.
  - **Integration**: Detects code changes and executes the full CI/CD pipeline.
- **6. Containerization & Deployment**
  - **Tools**: Docker, Kubernetes, Ansible
  - **Purpose**: Package apps into containers and deploy them across environments.
  - **Integration**: Jenkins can push Docker images to a container registry and trigger Kubernetes for deployment.
- **7. Monitoring & Feedback**
  - **Tools**: Prometheus, Grafana, ELK Stack, Nagios
  - **Purpose**: Monitor application health, performance, and log data.
  - **Integration**: Sends real-time alerts and feedback to developers for fixes or improvements.



PYQ. Illustrate how Jenkins is integrated with various devOps stages?

Ans=>

1. Plan

- **Tools:** Jira, Confluence, Trello (used outside Jenkins)
- **Jenkins Role:** No direct role, but Jenkins jobs may be triggered based on planning tickets via plugins/integrations with Jira.

2. Develop

- **Tools:** Git, GitHub, GitLab
- **Jenkins Integration:**
    - Jenkins polls the Git repository or uses webhooks to detect code changes.
    - Triggers **build jobs** automatically when a developer pushes code.

3. Build

- **Tools:** Maven, Gradle, Ant
- **Jenkins Role:**
    - Jenkins executes build scripts (e.g., mvn install) via build jobs.
    - Verifies the code compiles and dependencies are correctly managed.

4. Test

- **Tools:** Selenium, JUnit, TestNG, Postman
- **Jenkins Integration:**
    - Automates **unit tests**, **integration tests**, and **UI tests**.
    - Generates test reports and logs failures for review.

5. Release

- **Tools:** Docker, Nexus, Artifactory
- **Jenkins Role:**
    - Packages code into artifacts (e.g., JARs, Docker images).
    - Publishes to artifact repositories or container registries.

6. Deploy

- **Tools:** Kubernetes, Docker, Ansible, Terraform
- **Jenkins Integration:**
    - Automates deployment to test, staging, or production environments.
    - Can use Jenkins Pipelines (Declarative or Scripted) for step-wise deployment logic.

7. Monitor

- **Tools:** Prometheus, Grafana, New Relic

- **Jenkins Role:**
    - While Jenkins does not monitor applications, it can **trigger alerts**, integrate with monitoring APIs, or **record deployment success/failure** metrics.

Q7. Why DevOps become famous? How does AWS contribute to DevOps?

Ans=> **Why DevOps Became Famous:**

DevOps gained popularity due to the need for faster, more efficient software development and delivery. The traditional development and operations teams used to work in silos, leading to:

- **Slow deployments**
- **Lack of collaboration**
- **Frequent errors during handovers**
- **Long time to recover from failures**

DevOps solves this by:

1. **Faster Software Delivery**: Automation of build, test, and deployment processes.
2. **Improved Collaboration**: Dev and Ops teams work together using shared tools and processes.
3. **High-Quality Releases**: Continuous Integration/Continuous Deployment (CI/CD) ensures stable and tested code.
4. **Greater Agility**: Teams can respond quickly to changes and customer needs.
5. **Reduced Failures and Faster Recovery**: Automated testing and monitoring detect and fix issues early.

How AWS Contributes to DevOps:

AWS (Amazon Web Services) provides a wide range of tools and services that support every phase of the DevOps lifecycle:

DevOps Phase	AWS Contribution / Service
Source Code Management	AWS CodeCommit – Git-based source control
Build	AWS CodeBuild – Compiles code, runs tests, creates artifacts
Test	AWS Device Farm, third-party integrations (e.g., Selenium on EC2)
Release	AWS CodePipeline – Orchestrates release process
Deploy	AWS CodeDeploy – Automates deployments to EC2, Lambda, etc.
Operate	Amazon CloudWatch – Logs, metrics, monitoring
Monitor	AWS X-Ray, CloudTrail – Performance and user behavior tracking

Elaborate implementation of DevOps with an example.

📌 **Scenario:**

You're building an e-commerce website where you need continuous updates for features like user registration, product listings, cart, and payment integration. You want high availability, fast delivery, and minimal downtime.

🔧 **Step-by-Step DevOps Implementation:**

1. Plan

- **Tools:** Jira, Trello, Confluence
- **Goal:** Define user stories and requirements.
- **Example:** Product Manager creates a task "Add payment gateway".

2. Develop

- **Tools:** Git, GitHub/GitLab
- **Goal:** Developers write code collaboratively.
- **Example:** A developer creates a new feature branch add-payment-feature.

3. Build

- **Tools:** Maven, Gradle, Jenkins
- **Goal:** Compile and package code automatically.
- **Example:** Jenkins job runs mvn clean install whenever code is pushed.

4. Test

- **Tools:** JUnit, Selenium, Postman
- **Goal:** Automatically test the new code.
- **Example:** Jenkins runs unit and integration tests. Test reports are generated.

5. Release

- **Tools:** Jenkins, Nexus, Artifactory
- **Goal:** Store the build artifacts and make them ready for deployment.
- **Example:** .jar or Docker image is stored in Nexus after a successful build.

6. Deploy

- **Tools:** Docker, Kubernetes, AWS CodeDeploy
- **Goal:** Deploy code to testing or production environments.
- **Example:** Docker containers are deployed to a Kubernetes cluster on AWS.

7. Operate

- **Tools:** AWS CloudWatch, Prometheus, Grafana
- **Goal:** Monitor the application and infrastructure.

- **Example:** Real-time monitoring of user activity and system load.

8. Monitor & Feedback

- **Tools:** ELK Stack, New Relic
- **Goal:** Continuously monitor app performance and gather user feedback.
- **Example:** Alerts are sent to DevOps teams when payment service fails.

What is the GiT commit lifecycle, and what are its main stages?

✅ **Git Commit Lifecycle & Its Main Stages**

The **Git Commit Lifecycle** describes how files move through different stages before becoming part of the repository's history. Understanding this helps in effectively managing changes in a project.

🔁 **Lifecycle Overview:**

A file in Git typically passes through the following stages:

Stage	Description
1. **Untracked**	The file exists in your project directory but is not being tracked by Git.
2. **Tracked (Unmodified)**	The file is being tracked, and there are no changes since the last commit.
3. **Modified**	You've changed a tracked file but haven't staged it yet.
4. **Staged (Index)**	You've marked a file to be included in the next commit using git add.
5. **Committed**	You've saved the staged changes to the Git history using git commit.

🔄 **Detailed Flow with Commands:**

1. **Untracked ➡️ Staged:**

   git add <filename>

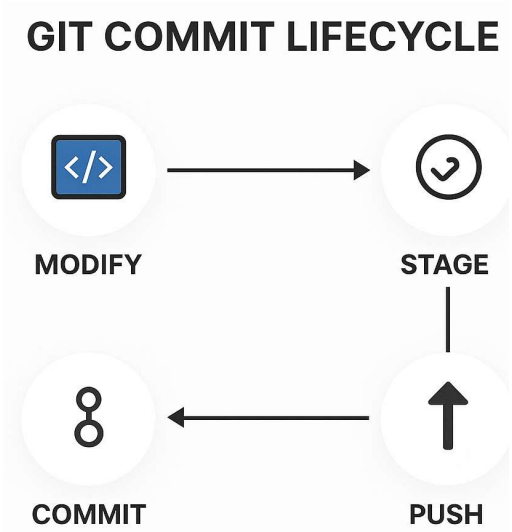
   - Tells Git to track the file and prepare it for commit.
2. **Staged ➡️ Committed:**

git commit -m "Initial commit"

   - Records the snapshot into the Git history.
3. **Tracked ➡️ Modified:**
   - You edit the file. Git detects a change.
4. **Modified ➡️ Staged:**

   git add <filename>

   - Again adds the modified file to the staging area.



5. **Repeat Commit:**

git commit -m "Updated file"

How to configure Docker in Jenkins?

How to Configure Docker in Jenkins on Windows

Configuring Docker in Jenkins on Windows involves several steps: installing Docker, installing Jenkins, integrating Docker with Jenkins, and verifying the setup.

1. Install Docker on Windows

- Download and install Docker Desktop for Windows from the official Docker website.
- Enable WSL 2 or Hyper-V, as required by Docker Desktop.
- After installation, verify Docker is running by executing docker --version in Command Prompt or PowerShell.

---

2. Install Jenkins

- Download and install Jenkins from the official Jenkins website.
- Install it as a Windows service or run it using a WAR file with java -jar jenkins.war.
- Access Jenkins at http://localhost:8080.

---

3. Install Required Jenkins Plugins

- Navigate to Manage Jenkins > Manage Plugins.
- Under the Available tab, install:
    - Docker Pipeline
    - Docker plugin
    - Docker Commons Plugin
- Restart Jenkins after plugin installation.

---

4. Configure Docker in Jenkins

- Go to Manage Jenkins > Global Tool Configuration.
- Under Docker installations, click Add Docker.
- Provide a name and set the Docker executable path (e.g., C:\Program Files\Docker\Docker\resources\bin\docker.exe or use docker if it's in system PATH).
- Save the configuration.

---

5. Set Up Permissions

- Ensure Jenkins has permission to access Docker:
    - If Jenkins runs as a Windows service, grant it proper permissions or run Jenkins under a user with Docker access.
    - Add the Jenkins user to the Docker group, or run Jenkins with administrator rights.

---

6. Verify the Setup

- Create a Freestyle or Pipeline job in Jenkins.
- Add a build step that runs a Docker command, like:

```
docker run hello-world
```

- Run the job to test Docker integration.

---

7. Use Docker in Jenkins Pipelines

- Create a Jenkinsfile:

```
pipeline {
    agent any
    stages {
        stage('Run Docker') {
            steps {
                script {
                    docker.image('alpine').inside {
                        sh 'echo Hello from inside Docker!'
                    }
                }
            }
        }
    }
}
```

DevOps Final PYQ

12Marks Questions

Create a Git workflow suitable for a small project team. How would you use Git commands to create a branch, make changes, and merge those changes into the main branch?

Ans=>

✅ **Git Workflow for a Small Team**

1. Main Branches

- **main**: The production-ready code. Always deployable.
- **develop**: The integration branch for features before they're merged to main.

2. Supporting Branches

- **feature/<name>**: For new features.
- **bugfix/<name>**: For minor bug fixes.
- **hotfix/<name>**: For critical fixes to production.
- **release/<version>**: For preparing releases.

3. Workflow Steps

Step 1: Clone the repo

git clone <repo_url>

cd <repo_name>

git checkout develop

Step 2: Create a feature branch

git checkout -b feature/login-form

Step 3: Work and commit locally

git add .

git commit -m "Add login form layout"

Step 4: Push to remote and open a Pull Request (PR)

git push origin feature/login-form

- Open a PR into develop.
- At least one teammate reviews and approves.

Step 5: Merge into develop

- Use **Squash and Merge** (recommended for cleaner history).

- Delete the feature branch.

Step 6: Prepare a release (if needed)

git checkout -b release/1.0.0 develop

Do final testing, version bumps, etc.

Step 7: Merge into main and tag

git checkout main

git merge release/1.0.0

git tag -a v1.0.0 -m "Release 1.0.0"

git push origin main --tags

Step 8: Merge back into develop

git checkout develop

git merge main

Step 9: Hotfixes (if production needs immediate patch)

git checkout -b hotfix/crash-fix main

Fix the issue, then...

git commit -am "Fix crash on startup"

git push origin hotfix/crash-fix

- Merge into main and tag a hotfix version (e.g., v1.0.1)
- Merge into develop as well.

To create a branch, make changes, and merge them into the main branch using Git, follow these steps:

1. Create and Switch to a New Branch

git checkout -b feature/my-feature

This creates a new branch called feature/my-feature and switches to it.

2. Make Changes and Commit

Make your code changes, then stage and commit them:

git add .

git commit -m "Add new feature"

3. Switch Back to the Main Branch

git checkout main

4. Merge the Feature Branch into Main

git merge feature/my-feature

5. Push the Updated Main Branch to Remote

git push origin main

6. (Optional) Delete the Feature Branch
git branch -d feature/my-feature

git push origin --delete feature/my-feature

Describe the concept of DevOps and explain its basic functionality.
Ans=>

🔹 **1. What is DevOps? (Concept)**

DevOps is a software development methodology that emphasizes **collaboration**, **automation**, and **integration** between **development (Dev)** and **operations (Ops)** teams. The goal is to **deliver software faster, more reliably, and with better quality** by breaking down silos between these traditionally separate functions.

DevOps is not just a set of tools, but a **culture and mindset** that promotes:

- **Shared responsibility** for the application lifecycle.
- **Continuous feedback** from end users and systems.
- **Faster iteration cycles** through automation.

🔹 **2. Key Principles of DevOps**

1. **Collaboration**: Dev and Ops teams work together throughout the lifecycle.
2. **Automation**: Repetitive tasks (e.g., testing, deployment) are automated.
3. **Continuous Integration and Delivery (CI/CD)**: Code is integrated, tested, and released more frequently.
4. **Monitoring and Feedback**: Systems are monitored in real-time, and feedback is used to improve code and performance.
5. **Infrastructure as Code (IaC)**: Infrastructure is provisioned and managed using code/scripts.

🔹 **3. Basic Functionalities of DevOps**

Below are the core functionalities that make DevOps effective:

🔸 **1. Continuous Integration (CI)**

- Developers merge code changes frequently into a shared repository.
- Automated tests run with each commit to ensure code stability.
- Helps detect bugs early in the development cycle.

Tools: Jenkins, GitLab CI, CircleCI

- **2. Continuous Delivery (CD)**
  - Builds on CI by automating the release process.
  - Code is automatically prepared for production after passing tests.
  - Reduces time to market.

Tools: GitHub Actions, AWS CodePipeline, Spinnaker

- **3. Infrastructure as Code (IaC)**
  - Servers, networks, and databases are managed using code.
  - Ensures consistent environments (e.g., staging matches production).
  - Enables version control for infrastructure.

Tools: Terraform, Ansible, CloudFormation

- **4. Automated Testing**
  - Includes unit tests, integration tests, and acceptance tests.
  - Ensures code quality and reduces human error.
  - Run automatically during the CI/CD pipeline.

Tools: Selenium, JUnit, TestNG

- **5. Configuration Management**
  - Automates software and system configuration across environments.
  - Makes systems predictable and easy to replicate.

Tools: Puppet, Chef, Ansible

- **6. Monitoring and Logging**
  - Real-time monitoring of applications and infrastructure.
  - Logs provide insight into errors and performance issues.
  - Enables quick detection and resolution of incidents.

Tools: Prometheus, ELK Stack (Elasticsearch, Logstash, Kibana), Grafana

- **7. Communication and Collaboration**
  - Encourages shared responsibility, faster feedback loops, and better teamwork.
  - Often supported by tools that integrate source control, task tracking, and alerts.

Tools: Slack, Jira, Confluence, Git

Explain the concept of Jenkins and also explain the initial installation and configuration of jenkins in detail. How Jenkins can be used to automate the build process of a software project. Also describe the process of linking Jenkins to a Git repository.
Ans=>

🔷 1. What is Jenkins? (Concept)

Jenkins is a popular **open-source automation server** used for **Continuous Integration (CI)** and **Continuous Delivery (CD)**. It helps in automatically building, testing, and deploying software projects.

🔹 Key Features:

- Written in Java, runs on any OS.
- Automates repetitive tasks in software development.
- Integrates with tools like Git, Maven, Docker, etc.
- Offers a web-based GUI and plugin support.

🔷 2. Jenkins Installation and Configuration on Windows

✅ Step-by-Step Installation:

Step 1: Install Java (JDK)

Jenkins requires Java to run.

1. Download JDK from https://www.oracle.com/java/technologies/javase-downloads.html
2. Install and set JAVA_HOME:
    - Right-click **This PC → Properties → Advanced system settings → Environment Variables**
    - Add new variable:
        - Name: JAVA_HOME
        - Value: C:\Program Files\Java\jdk-XX
3. Add JAVA_HOME\bin to the system **Path**.
4. Verify in Command Prompt:

```
java -version
```

Step 2: Download and Install Jenkins

1. Go to the Jenkins website:
   👉 https://www.jenkins.io/download/
2. Download the **Windows installer (.msi)** version.
3. Run the installer:
    - Jenkins will install as a **Windows service**.
    - By default, it runs on: http://localhost:8080

Step 3: Unlock Jenkins (First Time Setup)

1. Open browser:
   http://localhost:8080
2. It will ask for an **admin password**.
    - Find it here:

C:\Program Files (x86)\Jenkins\secrets\initialAdminPassword

3. Paste the password into the browser and click **Continue**.

Step 4: Install Plugins and Setup Admin User

1. Choose **"Install Suggested Plugins"** (recommended).
2. Create a new **Admin User** (username, password, email).
3. Jenkins is now ready to use.

🔷 **3. Automating Build Process in Jenkins**

✅ **Jenkins automates builds using Jobs (Projects):**

Step-by-Step Process:

1. **Create a New Job:**
    - Click on **"New Item"**
    - Give it a name (e.g., BuildJavaApp)
    - Choose **Freestyle project** → Click OK
2. **Configure Source Code:**
    - If you're using Git, enter the repo URL under **Source Code Management**.
3. **Add Build Trigger:**
    - You can use:
        - **Poll SCM** – Jenkins checks Git repo for changes (e.g., H/5 * * * *)
        - **Build periodically** – Scheduled builds

4. **Add Build Step:**
    - Example using Maven:

```
mvn clean install
```

    - Or add a Windows batch command:

```
javac HelloWorld.java
java HelloWorld
```

5. **Post-build Actions (Optional):**
    - Send email notifications
    - Archive build artifacts
    - Deploy to a web server or cloud
6. **Save and Build:**
    - Click **Build Now**
    - View logs via **Console Output**

🔷 4. Linking Jenkins to a Git Repository (in Detail)

✅ Step-by-Step Git Integration:

1. **Install Git on Windows:**
    - Download Git from https://git-scm.com
    - Install and make sure it's available in the system PATH.
    - Verify:

```
git --version
```

2. **Install Git Plugin in Jenkins (if not preinstalled):**
    - Go to **Manage Jenkins → Plugins → Available**
    - Search for **Git Plugin** → Install
3. **Configure Git in Jenkins Job:**

- In the Job configuration:
    - Under **Source Code Management**, select **Git**
    - Enter your Git repository URL:

```
https://github.com/username/repository-name.git
```

4. **Add Git Credentials (if private repo):**
    - Click **Add Credentials** under the Git section

- Choose **Username & Password** or **SSH Key**

5. **Add Build Trigger (optional):**
    - Enable **Poll SCM** to check for new commits.
6. **Build Steps:**
    - Jenkins will now clone the repo, compile code, and run tests automatically.

Illustrate the concept of DevOps. Explain how it works? Also explain various DevOps tools in detail.

🔷 1. What is DevOps?

DevOps is a software development methodology that emphasizes collaboration between **development (Dev)** and **operations (Ops)** teams to automate and improve the process of software delivery and infrastructure changes. It aims to bridge the gap between development and operations by fostering a culture of **collaboration, communication, and shared responsibility**.

Key Features of DevOps:

- **Collaboration**: Breaks down silos between development and operations teams.
- **Automation**: Automates the entire software delivery lifecycle, including testing, deployment, and monitoring.
- **Continuous Integration and Continuous Delivery (CI/CD)**: Integrates and deploys code frequently and automatically.
- **Infrastructure as Code (IaC)**: Manages infrastructure using code, making it easy to replicate and scale.
- **Monitoring and Feedback**: Provides real-time feedback to improve systems and applications.

🔷 3. How DevOps Works?

DevOps works by integrating different processes and tools across the software development lifecycle, automating manual processes, and ensuring that teams collaborate efficiently. Here's how DevOps typically works in practice:

🔸 1. Plan and Develop

- The development team writes code and creates features in short iterations.
- **DevOps Tools**: Jira for task tracking, Git for version control, and collaboration platforms like Confluence.

🔸 2. Continuous Integration (CI)

- Developers commit code to a **shared repository** frequently (e.g., GitHub, GitLab).
- Each commit triggers an automated **build and test process** using tools like **Jenkins, CircleCI, or TravisCI**.

- Automated tests (unit tests, integration tests) are run to ensure code quality.

Key Tools:

  - Jenkins, GitLab CI, CircleCI, Travis CI.

🔸 **3. Continuous Delivery (CD)**

- Once the code passes tests, it is automatically **deployed** to **staging** or **production**.
- **Deployment Automation** tools such as **Ansible**, **Chef**, or **Docker** are used for managing environments.
- The code is continuously integrated, tested, and deployed, allowing new features or updates to be available to customers faster.

Key Tools:

  - Kubernetes (for orchestration), Docker (for containerization), Ansible (for automation), Terraform (for infrastructure provisioning).

🔸 **4. Monitor and Operate**

- After deployment, DevOps ensures that systems are **monitored** continuously for issues and performance metrics.
- Tools like **Prometheus**, **Grafana**, **Nagios**, and **ELK Stack (Elasticsearch, Logstash, Kibana)** are used to monitor infrastructure and applications in real-time.
- **Feedback loops** are established, allowing teams to fix issues proactively based on user behavior and system performance.

Key Tools:

  - Prometheus, Grafana, ELK Stack, Nagios.

🔸 **5. Continuous Feedback**

- Monitoring tools provide real-time data, and developers continuously improve the code based on user feedback.
- Feedback loops ensure that any operational issues or bugs are immediately identified and fixed, fostering continuous improvement.

Compare and contrast the process and benefits of creating Docker images versus using containers. Also describe the layers of a Docker image and their significance. Explain in detail the process Of pulling a Docker image from Docker Hub and pushing a locally created Docker image to your Docker Hub repository. Include the specific commands.

1. Docker Images vs. Docker Containers: Process and Benefits

A. Docker Image

A **Docker image** is a **read-only** template that defines the **file system** and the **instructions** needed to run a containerized application. Docker images are **immutable**, which means they cannot be changed once created. However, they serve as the blueprint from which containers are created.

Process of Creating Docker Images:

- **Build a Docker Image**: You create a Docker image using a file called Dockerfile. The Dockerfile contains instructions (commands) to build the image. Common instructions include setting the base image, installing dependencies, copying files, and setting the default command to run in the container.
- **Image Creation**: After writing a Dockerfile, you run the docker build command to create an image.

Benefits of Docker Images:

1. **Reusability**: Images can be shared and reused across environments, ensuring consistency.
2. **Portability**: Images contain everything needed to run the application, making it easy to move between environments (development, testing, production).
3. **Version Control**: Docker images can be tagged with version numbers, making it easy to manage and roll back to previous versions.
4. **Efficiency**: A single image can be used to spin up multiple containers on different hosts.

B. Docker Container

A **Docker container** is a **running instance** of a Docker image. Containers are **mutable**, meaning changes can be made while running, but they are ephemeral (temporary). Once a container is stopped, it can be removed, and any changes made to it are lost unless they are committed back to an image.

Process of Using Docker Containers:

- **Run a Docker Container**: You can create and start a container from an image using the docker run command. The container has its own file system, process space, and network interfaces, but it shares the kernel of the host system.
- **Container Lifecycle**: A container runs the application inside the image, and when the container is stopped, it can be restarted, removed, or committed to create a new image.

Benefits of Docker Containers:

1. **Isolation**: Each container runs in its own isolated environment, preventing conflicts between applications.
2. **Portability**: Containers can run on any machine that has Docker installed, regardless of the underlying operating system.

3. **Resource Efficiency**: Containers share the host system's kernel, making them lightweight and more efficient than traditional virtual machines.
4. **Fast Deployment**: Containers start and stop quickly, making them ideal for microservices and rapid development cycles.

2. Layers of a Docker Image and Their Significance

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 as installing software, copying files, or setting environment variables.

Docker Image Layers:

1. **Base Layer (or Base Image)**:
    - This is the first layer of the image and is usually the OS or minimal environment (e.g., ubuntu, alpine).
    - 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) adds a new layer to the image.
    - For example, RUN apt-get install -y python creates a new layer with the Python installation.
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, defined by the CMD or ENTRYPOINT instruction in the Dockerfile.

3. Pulling a Docker Image from Docker Hub

Docker Hub is the default registry where Docker images are stored. You can pull an image from Docker Hub using the docker pull command.

Steps to Pull a Docker Image:

1. **Search for the image** (optional):

docker search <image-name>

Example: docker search ubuntu

2. **Pull the image**:

docker pull <image-name>

For example, to pull the latest version of the official Ubuntu image:

docker pull ubuntu

3. **Verify the pull**:
   After pulling the image, you can verify that it has been downloaded by running:

docker images

4. Pushing a Docker Image to Docker Hub

After creating or modifying a Docker image locally, you can push it to Docker Hub so that it can be shared and used by others.

Steps to Push a Docker Image to Docker Hub:

1. **Create a Docker Hub account**:
   - Go to https://hub.docker.com/ and sign up for an account.
2. **Log in to Docker Hub** (using Docker CLI):

docker login

Enter your Docker Hub username and password.

3. **Tag the Docker Image**:
   Docker requires that images be tagged with a name that includes your Docker Hub username and a repository name. The format is username/repository:tag.

For example, if your Docker Hub username is yourusername and you want to push an image named myapp, you would tag your image as follows:

docker tag <local-image> yourusername/myapp:latest

Example:

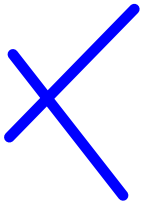
docker tag myapp yourusername/myapp:latest

4. **Push the Image**:
   After tagging the image, you can push it to your Docker Hub repository:

docker push yourusername/myapp:latest

5. **Verify the Push**:
   After the push, you can verify the image is available in your Docker Hub account by visiting your repository on Docker Hub or by using the Docker CLI:

docker pull yourusername/myapp:latest



Detail the working with Bourne Again Shell (BASH) using GUL Explain the GIT installation steps.

Ans=>

Introduction to BASH (Bourne Again SHell)

BASH is a command-line interpreter for the GNU operating system. It's a **Unix shell** and command language that allows users to interact with the system via terminal or scripts. While BASH is a command-line tool, you can interact with it using **GUI-based terminal emulators** like:

- **Git Bash for Windows** (GUI emulator with BASH capabilities)
- **Gnome Terminal**, **Konsole**, or **Terminator** on Linux
- **Terminal.app** or **iTerm2** on macOS

2. Working with BASH (Using Git Bash - GUI Tool)

Git Bash is a GUI-based terminal emulator for Windows that lets you use BASH commands in a graphical environment.

Basic BASH Commands in Git Bash (GUI):

Command	Description
ls	List files in the directory
cd foldername	Change directory
mkdir foldername	Create a new folder
touch filename	Create a new empty file
rm filename	Delete a file
git init	Initialize a Git repository
git clone <url>	Clone a Git repo
git status	Show file changes
git add .	Stage all changes
git commit -m "message"	Commit changes
git push origin main	Push code to the repository

Example Workflow Using Git Bash (GUI):

mkdir project

cd project

git init

touch index.html

git add index.html

git commit -m "Initial commit"

These steps can all be done using Git Bash, which opens as a GUI terminal in Windows.

3. Git Installation Steps (Windows, Linux, macOS)

A. On Windows:

Step-by-Step Installation of Git:

1. **Download Git for Windows:**
    - Go to: https://git-scm.com/download/win
2. **Run the Installer:**
    - Double-click the downloaded .exe file.
3. **Choose Setup Options:**
    - Choose components to install. Keep defaults unless you have specific needs.
    - Select a text editor (Notepad++, VS Code, etc.)
    - Choose “Use Git from the command line and also from 3rd-party software” (recommended).
4. **Configure Line Endings:**
    - Choose **"Checkout Windows-style, commit Unix-style line endings"** (recommended).
5. **Complete Installation:**
    - Click "Install" and wait for the process to finish.
6. **Verify Installation:**
    - Open **Git Bash** and type:

```
git --version
```