**1. Introduction to CI/CD**

**• Definition:** CI/CD stands for Continuous Integration and Continuous Delivery.

**◦ Continuous Integration (CI):** This is a process where you integrate a set of tools or follow a set of processes before delivering your application to the customer. CI ensures that your build is smooth, all tests are executed, code quality is maintained, and an image (like a Docker image) is created.

**◦ Continuous Delivery (CD):** This is the process where you deploy or deliver your application to a specific platform for your customer. CD ensures that your deployment or delivery process is done efficiently and reliably.

**• Purpose:** CI/CD automates the various steps involved in getting an application from development to the customer, drastically reducing the time it takes from months to weeks or even days. It ensures applications are tested, scanned for security vulnerabilities, and perfect before deployment.

**• Benefits: I**mproves efficiency, ensures reliability, automates repetitive tasks, and accelerates application delivery. Without CI/CD, applications would take months to deliver.

**2. CI/CD Workflow:** From Development to Production

The entire workflow involves promoting an application through various environments.

**• Code Commit and Version Control:**

◦ A developer makes code changes, typically to a feature branch in a Version Control System (VCS) like GitHub.

◦ Organizations use VCS (e.g., GitHub, Bitbucket, GitLab) to store code and track changes.

**• Triggering the CI Pipeline:**

◦ Once a developer commits code to the repository (e.g., a feature branch), a webhook on the GitHub repository can trigger a Jenkins pipeline.

◦ Webhooks are the most efficient way to trigger pipelines because they send a notification to Jenkins only when a specific action occurs, avoiding constant polling by Jenkins. Alternatives like polling (Jenkins continuously checking the repository) or build triggers are less efficient and costly due to continuous API calls.

**• Continuous Integration (CI) Pipeline Stages (e.g., in Jenkins):**

**◦ Orchestrator:** Jenkins acts as an orchestrator, or a pipeline/tunnel, automating and integrating various tools.

**◦ Code Checkout:** The Jenkins pipeline first checks out the code from the Git repository.

**◦ Building and Unit Testing:** The application is built (e.g., using Maven for Java, npm for Node.js) and unit tests are run.

**◦ Code Scanning/Quality Analysis:** Tools like SonarQube perform static code analysis to check for static code, duplicate code, linting issues, syntax, and semantic issues.

**◦ Image Build Creation:** For containerized applications (microservices), a container image is created using tools like Docker, Buildah, or Podman. Docker is a popular choice.

**◦ Image Scanning:** The newly created container image is scanned for critical vulnerabilities using tools like Trivy, Snyk, or Clair.

**◦ Pushing Image to Registry:** The final container image is pushed to an image registry (e.g., Docker Hub, Quay, Artifactory, ECR).

**• Updating Kubernetes Manifests:** After a new image is created and pushed, the Kubernetes manifests (e.g., deployment.yaml, StatefulSet, DaemonSet, or Helm charts) are updated with the new image version. It's good practice to keep these manifests in a separate Git repository (a "manifest repository").

**3. Branching Strategy and Application Promotion**

Understanding the branching strategy is key to the entire promotion process.

**• Types of Branches:**

◦ **Main Branch (formerly Master Branch):** This is the single branch where active development is continuously going on. It contains reliable code that has been integrated with all other changes.

**◦ Feature Branches:** Developers create these for new enhancements or features. There can be many feature branches, each for a specific piece of work.

**◦ Release Branches:** Created from the main branch when features are ready for a release to production. Different versions of an application in production originate from release branches.

**◦ Hotfix Branches:** Created from a specific release branch (or even directly from production code) to fix critical bugs found in production. Changes from a hotfix branch are then merged back into the release, main, and feature branches.

**• Environment Promotion Workflow:**

**◦ Development Environment (Feature Branch Environment):** Changes committed to a feature branch are deployed to a development Kubernetes cluster. This environment is specific to a particular feature a developer is working on.

**◦ Staging Environment (QA Environment):** Once feature branch changes are merged into the main branch, they are deployed to a staging environment (or QA environment) on a different Kubernetes cluster. The QA team performs additional manual tests (performance, penetration testing) here over a few days.

**◦ Pre-production Environment (UAT Environment):** After QA sign-off on the staging environment, a release branch is created from the main branch. A release pipeline (Jenkins pipeline) is triggered, deploying to a pre-production environment (or UAT environment). This environment is used for debugging production issues and reproducing bugs without impacting live systems.

**◦ Production Environment:** If everything looks good in pre-production, the application is promoted from the same release pipeline to the production environment. This is the exact environment customers use, typically with a robust setup.

**◦ Different Kubernetes Clusters:** It's a good practice to have separate Kubernetes clusters for development, staging, pre-production, and production environments, though sometimes namespaces within a cluster are used for dev/staging (but not for pre-prod/prod).

**4. Jenkins Deep Dive**

**• Jenkins as an Orchestrator:** Jenkins orchestrates the various tools and steps in the CI/CD pipeline.

**• Jenkins Pipelines (Pipeline as Code):**

◦ Jenkins pipelines define the CI/CD workflow as code, typically written in Groovy scripting.

◦ Declarative Pipelines are preferred over Scripted Pipelines because they are easier to write, understand, and collaborate on, especially for teams less experienced with extensive programming.

◦ Jenkins files (the pipeline code) can be stored in the Git repository alongside the source code. This allows for version control, peer reviews, and easy tracking of changes.

◦ The Pipeline Syntax generator in Jenkins UI helps in generating Groovy script snippets for various operations (e.g., Git checkout, shell execution).

**• Jenkins Agents (Worker Nodes):**

**◦ Traditional Worker Nodes (VMs/EC2 Instances):** Jenkins Master offloads build tasks to worker nodes to handle load and dependency conflicts (e.g., different Java/Python versions for different teams). However, this approach can lead to idle resources (wasting compute), high maintenance overhead (upgrading dependencies, OS), and scaling challenges.

**◦ Docker Agents:** The modern and highly recommended approach is to use Docker containers as Jenkins agents.

**▪ Advantages:**

• **Lightweight and Efficient:** Containers are much lighter than VMs.

• **Cost-Saving:** Containers are created dynamically only when a job runs and are deleted immediately after execution, preventing idle resource wastage.

• **Simplified Dependency Management:** Each container can have its specific set of tools and dependencies (e.g., one for Java/Maven, another for Node.js), avoiding conflicts on a single machine.

• **Reduced Maintenance Overhead:** Updating dependencies simply involves changing the Docker image tag in the Jenkinsfile, rather than manually updating multiple VMs.

• **Scalability:** Easily scales up and down.

**• Shared Libraries:**

◦ **Concept:** Shared libraries allow you to centralize repetitive or reusable Groovy code (functions, stages) from your Jenkinsfiles into a separate Git repository. This promotes "low code" practices in pipelines.

**◦ Advantages:**

**▪ Standardization:** Ensures consistent pipeline practices across teams/projects.

**▪ Reduced Code Duplication:** Avoids copying the same code across hundreds of Jenkins pipelines.

▪ **Easier Maintenance:** A single change in the shared library is reflected across all pipelines that use it, significantly reducing effort for updates or security fixes.

▪ **Faster Onboarding**: New applications or team members can quickly set up pipelines by simply referencing shared libraries.

**▪ Reduced Risk of Errors:** Less code to write means fewer potential errors.

**◦ Implementation:** Create a vars folder in a Git repository (your shared library repo). Inside, define Groovy files (e.g., mavenBuild.groovy) with a def call() function containing the reusable code. Configure Jenkins to recognize this shared library repository, then import and call the library methods in your Jenkinsfiles.

**• Jenkins Backup and Scaling:**

**◦ Backup:** The core Jenkins configuration and data are stored in the .jenkins folder (a hidden folder). You can back this up periodically using tools like rsync to an EBS volume or snapshots. Some organizations also use external databases for Jenkins data, which also need to be backed up.

**◦ Scaling:** Jenkins can be scaled up or down using Auto Scaling Groups (e.g., on AWS EC2 instances). Predictive scaling can automatically add/remove worker nodes based on anticipated load, with Jenkins using SSH to connect to new nodes.

• **Plugins:** Essential for extending Jenkins functionality. Examples include Docker Pipeline Plugin (for Docker agents) and Sonar Scanner Plugin (for SonarQube integration). Plugins can be installed via the Jenkins UI or CLI.

**• Secrets Management:**

◦ It's critical to secure sensitive information (API keys, credentials, SSH keys) to prevent leaks in logs or UI.

**◦ Jenkins Credential Plugins:** Jenkins has built-in features to store credentials.

◦ Hashicorp Vault: A widely used, robust tool for securing and storing secrets, integrable with Jenkins and other DevOps tools.

**◦ Cloud-specific solutions:** AWS Systems Manager, Azure Vault

**5. Continuous Delivery (CD) Tools: GitOps**

**• GitOps Approach:** This is a modern, declarative continuous delivery model. Instead of Jenkins directly deploying to Kubernetes, Argo CD (or similar tools) monitors a Git repository for application manifests.

**◦ Two Git Repositories:** Typically, there's a source code repository and a manifest repository (for Kubernetes yaml files or Helm charts).

**◦ Pull Mechanism:** Unlike CI (which is push-based from Git to Jenkins), CD with GitOps is a pull mechanism. The CD tool pulls changes from the manifest repository.

◦ Advantages of GitOps for CD:

**▪ Single Source of Truth:** Git is the single source of truth for your application's desired state.

**▪ Declarative:** Describes the desired state, and the tool ensures the cluster matches it.

▪ **Automated Sync:** Continuously monitors the Git repository for changes and automatically deploys them to the Kubernetes cluster.

▪ **Drift Detection and Self-Healing:** GitOps tools (Kubernetes controllers) can detect configuration drift (manual changes made directly to the cluster) and automatically revert them to match the Git repository's state, ensuring consistency and preventing unauthorized changes.

**▪ Auditing and Rollback:** All changes are version-controlled in Git, enabling clear auditing and easy rollbacks by simply reverting Git commits.

◦ Why not traditional CD (e.g., Ansible/Shell Scripts from Jenkins CI)?: Traditional methods are often not scalable, not designed for continuous delivery, and lack the robust state management and drift detection capabilities of GitOps tools.

**• Argo CD:**

◦ A popular declarative continuous delivery tool (Kubernetes controller) that runs inside your Kubernetes cluster.

◦ Watches the Git manifest repository for changes (e.g., new image tags in deployment.yaml).

◦ Automatically deploys or syncs these changes to the target Kubernetes cluster.

**• Argo Image Updater:**

◦ A tool that monitors container registries (like Docker Hub or ECR).

◦ When a new image version is pushed to the registry, Argo Image Updater automatically updates the image tag in your Git manifest repository (e.g., in a deployment.yaml or Helm values.yaml), creating a new commit. This commit then triggers Argo CD to deploy the new version.

**• Deployment Strategies (for CD):**

**◦ Blue-Green Deployment:** Deploying a new version (green) alongside the existing one (blue). Traffic is then shifted from blue to green via a load balancer. If issues arise, traffic can be immediately shifted back to the stable blue version, providing fast rollbacks.

**◦ Canary Deployment:** Gradually shifting a small percentage of traffic to the new version (canary) while the old version serves most users. This allows for testing in a live environment with minimal impact, and if issues are detected, traffic can be reverted or gradually increased.

**Analogy for CI/CD Workflow:** Think of CI/CD like baking a perfect cake in a highly automated bakery.

**• Developers adding ingredients (code changes):** This is like a chef adding new ingredients or modifying existing ones for a recipe.

**• Git Repository (Recipe Book):** All the recipe variations and ingredients lists are stored here, version-controlled.

**• CI Pipeline (Automated Mixer, Oven, Quality Control):** When a new recipe variation is added, the automated system (Jenkins) kicks in.

◦ It pulls the recipe (code checkout).

◦ It mixes the ingredients (build and unit test), ensuring the batter is smooth and correct.

◦ It scans the batter for impurities or allergens (code scanning).

◦ It bakes the cake in a perfectly shaped mold (Docker image build).

◦ It quickly scans the baked cake for any cracks or defects (image scanning).

◦ Finally, it places the cake in a refrigerated display case (image registry).

**• Branching Strategy (Recipe Categories & Batches):** Different recipe variations (feature branches), the master recipe (main branch), seasonal recipes (release branches), and emergency fixes for a spoiled batch (hotfix branches).

**• CD Pipeline (Automated Delivery Truck, Smart Inventory System)**:

◦ The "Smart Inventory System" (Argo Image Updater) constantly monitors the display case for new cake versions.

◦ When a new cake version appears, it automatically updates the "Delivery Order" (manifest repository) with the new cake's ID.

◦ The "Automated Delivery Truck" (Argo CD) continuously checks the "Delivery Order." As soon as it sees a new cake ID, it pulls the cake from the display case and delivers it to the customer's plate (Kubernetes cluster).

◦ If a customer tries to modify their delivered cake (e.g., add frosting directly to the cake on their plate), the "Delivery Truck" notices it doesn't match the "Delivery Order" and automatically replaces it with the correct cake, ensuring the customer always gets the officially ordered version.