What is CICD?

CICD: An infinite loop diagram with icons for Plan, Code, Build, Test, Release, Deploy, Operate, Monitor]

- **The Definition:**
  - CI/CD stands for Continuous Integration and Continuous Delivery/Deployment.
  - It is a set of practices that automates the processes between software development and IT teams.
- **The Core Concept:**
  - Instead of releasing software once every few months (the "Waterfall" model), CI/CD allows teams to deliver code changes to users frequently, reliably, and automatically.
- **The Goal:**
  - To build a "Software Assembly Line" where code moves from the developer's laptop to the production server without manual bottlenecks.

CI (Continuous Integration)

Multiple code branches merging into a single main branch, triggering a robot icon

- Focus: Integrating code changes frequently.
- **How it works:**
  - Developers write code and push it to a shared repository (like GitLab) multiple times a day.
  - Each push triggers an automated build (compiling the code).
  - Automated tests (Unit tests) run immediately to verify the code.
- **The Benefit:**
  - "Fail Fast": If a developer introduces a bug, the team finds out within minutes, not weeks.
  - It eliminates "Integration Hell" (the nightmare of trying to combine weeks of conflicting code changes at the last minute).

CD (Continuous Delivery & Deployment)

A pipeline flow. Code passing through a gate labeled "Staging" and then automatically flowing to "Production"

- **Continuous Delivery (The "Staging" Phase):**
  - Code changes are automatically built, tested, and prepared for a release to a testing environment.
  - Key Point: The software is *always* ready to be deployed to production, but it requires a human approval (a button click) to go live.
- **Continuous Deployment (The "Production" Phase):**
  - This takes Delivery a step further. There is no human intervention.
  - If tests pass, the code is automatically deployed to the live environment immediately.
  - *Example:* Companies like Facebook and Netflix use Continuous Deployment to push updates thousands of times a day.

Real-World Problem CI/CD Solves

Scenario Without CI/CD

- Developer emails code
- Ops team manually deploys
- Application breaks in production

With CI/CD

- Developer pushes code
- Pipeline runs automatically
- Tested, packaged, deployed safely

CI/CD Tools

- GitHub Actions
- GitLab CI/CD
- Jenkins
- Azure DevOps

- Why GitLab CI/CD?
  - The "All-in-One" Advantage: GitLab is not just a CI tool; it is a complete DevOps platform (Project Management + Source Code + CI/CD + Monitoring) in a single application.
  - No Integration Headaches: You don't need to connect your Repo to a separate CI server; it's built-in.

CI/CD Pipeline Overview

What is a Pipeline?

- Automated steps executed on code changes

Pipeline Triggers

- Git push
- Pull request
- Merge to main branch

Pipeline Stages:

1. Build
2. Test
3. Package
4. Deploy

Pull Requests & Code Merge

- Developer creates Pull Request
- Code is reviewed
- CI pipeline runs automatically
- If successful → merge to main/develop

Understanding the Workflow

1. Developer pushes code to a branch (`feature/login`).
2. GitLab detects the `.gitlab-ci.yml` file.
3. GitLab creates a Pipeline.
4. The Pipeline is divided into Stages (e.g., Build, Test, Deploy).
5. Each Stage contains Jobs (e.g., `test_unit`, `test_integration`).
6. Runners execute the Jobs.
7. Result: Success (Green) or Failure (Red).

CI/CD Tools

- A hierarchy chart: Pipeline → Stage → Job

- Pipeline: The entire unit of work. It groups all jobs and stages for a specific commit.
- Stage: Defines *when* a job runs. Jobs in the same stage run in parallel. Stages run sequentially (e.g., Build must finish before Test starts).
- Job: The smallest unit. It defines a specific task (e.g., "Run npm test").
- Runner: The server/container that actually runs the shell commands defined in the job.
  - *Shared Runner:* Provided by GitLab (good for open source/small teams).
  - *Specific Runner:* Hosted by your company (good for security/private tools).

Deep Dive - The `.gitlab-ci.yml` File

- This file lives in the root of your repository.
- It defines the logic of your pipeline.

yaml

```yaml
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script: "echo 'Compiling the code...'"

test_job:
  stage: test
  script: "echo 'Running tests...'"

deploy_job:
  stage: deploy
  script: "echo 'Deploying to server...'"
  when: manual  # This requires a human click to start
```

Advance features

- Caching & Artifacts:
    - *Cache:* Speed up builds by keeping dependencies (like `node_modules`) between jobs.
    - *Artifacts:* Save the compiled output (like an `.exe` or `.apk`) to download later.
- Environments:
    - Track deployments to `Development`, `Staging` and `Production`. You can see exactly what version is running where.

GitLab CI/CD Architecture

GitLab Server ↔ GitLab Runner ↔ Your Code

- **The Core Components:**
  1. **GitLab Server:** The brain. It stores your code and detects changes (commits/pushes). It stores the pipeline configuration.
  2. **.gitlab-ci.yml:** The heart. A file in your repo that tells GitLab *what* to do.
  3. **GitLab Runner:** The muscle. A lightweight agent that picks up jobs from the GitLab server and executes the scripts (builds/tests).

Branch Strategy

Our backend repository uses multiple branches for managing development and deployment environments.

Main Branch Types:

1. **Production Branches**
   - Production
   - Production_v2
   - Production_v3
   - production_v4
   - temp_production
2. **Staging Branches**
   - Staging
   - Staging_v2
   - Staging_v3
   - Staging_v4
   - staging_email_changes
3. **Development / Feature Branches**
   - admin_fixes
   - fixing_data
   - forgot_password
   - phase_2_data
   - unsubscribe_email_token
   - refresh_time_update

Why Multiple Production Versions Exist

- Supports version-based releases
- Allows rollback to previous stable versions
- Helps manage hotfixes
- Supports parallel deployment testing
- Ensures minimal downtime

Why Multiple Staging Versions Exist

- Allows testing for different production versions
- Supports QA testing without affecting live users
- Helps test new features safely
- Ensures staging environment mirrors production

What is `.gitlab-ci.yml`?

`.gitlab-ci.yml` is the configuration file that:

- Defines CI/CD pipeline rules
- Defines stages of deployment
- Controls automation flow
- Runs build, test, and deployment jobs

Each branch contains its own `.gitlab-ci.yml` to manage environment-specific workflows.

Pipeline Stages

Typical CI/CD pipeline stages:

1. **Build Stage**
   - Installs dependencies
   - Compiles backend code
   - Creates build artifacts
2. **Test Stage**
   - Runs unit tests
   - Validates code quality
3. **Deploy Stage**
   - Deploys application to servers
   - Configures environment variables
   - Restarts services

Branch Wise Pipeline Flow

Staging Branch Pipeline Flow:

When code is pushed to staging branch:

1. Pipeline automatically triggers
2. Dependencies are installed
3. Build process starts
4. Tests are executed
5. Code deploys to staging server
6. QA team tests new features

Purpose:

- Validate functionality before production deployment

Branch Wise Pipeline Flow

Production Branch Pipeline Flow:

When code is pushed to production branch:

1. Pipeline triggers production deployment
2. Builds production-ready application
3. Uses production environment variables
4. Deploys code to live server
5. Application becomes available to end users

Purpose:

- Provide stable and tested releases

Trigger Rules Example

- Pipeline runs automatically on push
- Different branches trigger different deployment environments
- Manual approval may be required for production

Example:

- Push to staging → deploys to staging server
- Push to production → deploys to live server

Deployment Architecture

Deployment Flow

Code Flow:

Developer → Git Repository → GitLab Pipeline → Server Deployment → Application Running

Environment Variables

Environment variables help store:

- Database credentials
- API keys
- Server configuration
- Security tokens

Benefits:

- Improves security
- Avoids hardcoding sensitive data
- Allows environment-specific configuration

Deployment Architecture

Server Communication

- GitLab runner executes deployment scripts
- Server receives updated code
- Application services restart
- Logs and monitoring validate deployment