pipeline → sequence of stages

Integration

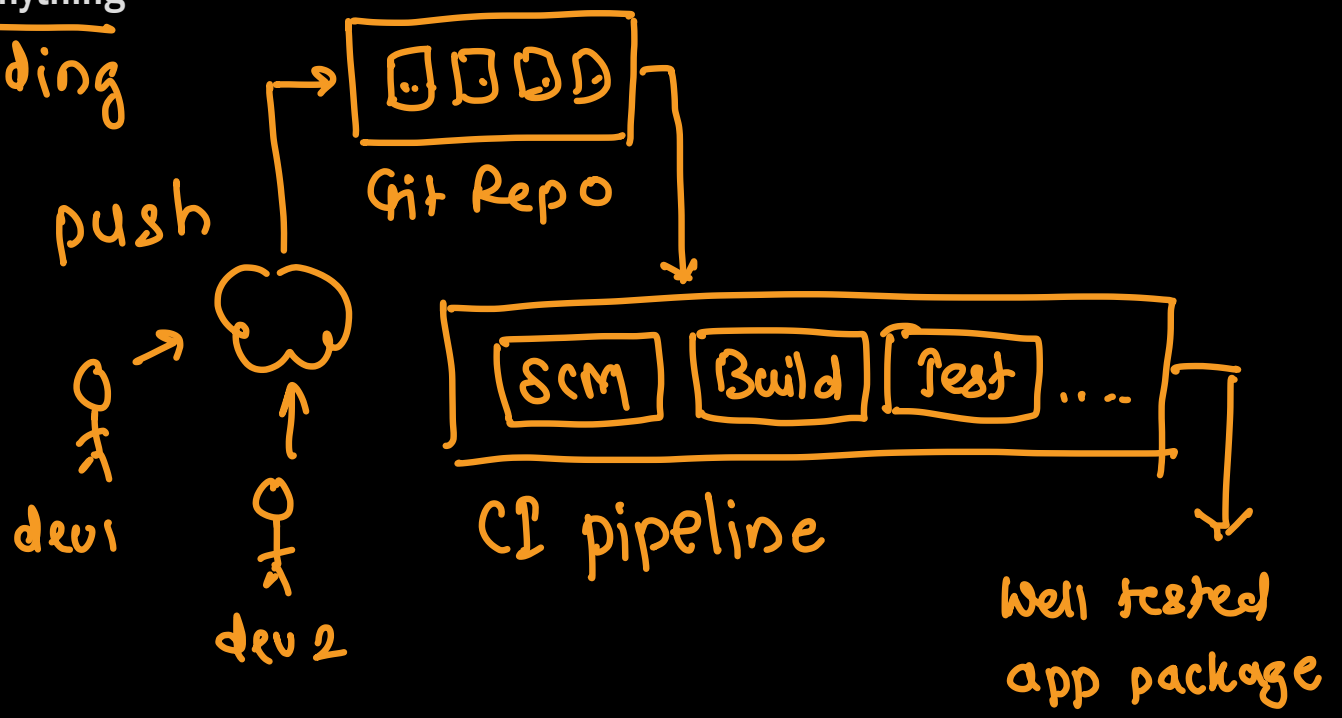
↓

CI/CD

- Delivery → manual
- Deployment → automatic

Continuous Integration (CI)

- This is the practice of automatically integrating code changes from multiple developers into a shared repository frequently—often several times a day
- **What happens:**
  - Developers push code to a version control system (like Git)
  - Automated tests run to check if the new code breaks anything
  - The code is automatically built/compiled → building
  - Teams get immediate feedback if something is wrong



The Problem CI Solves

- **Before CI**
  - Developers worked on separate branches for weeks or months
  - When merging, massive conflicts occurred ("integration hell")
  - Bugs were discovered late, making them expensive to fix
  - No one knew if the combined code actually worked until the end
  - Building and testing were manual, slow, and error-prone
- **With CI** (automation)
  - Small, frequent integrations reduce conflict complexity
  - Problems are detected within minutes, not weeks
  - The codebase is always in a working state
  - Team collaboration improves with better visibility

feature branch

git → master / main branch

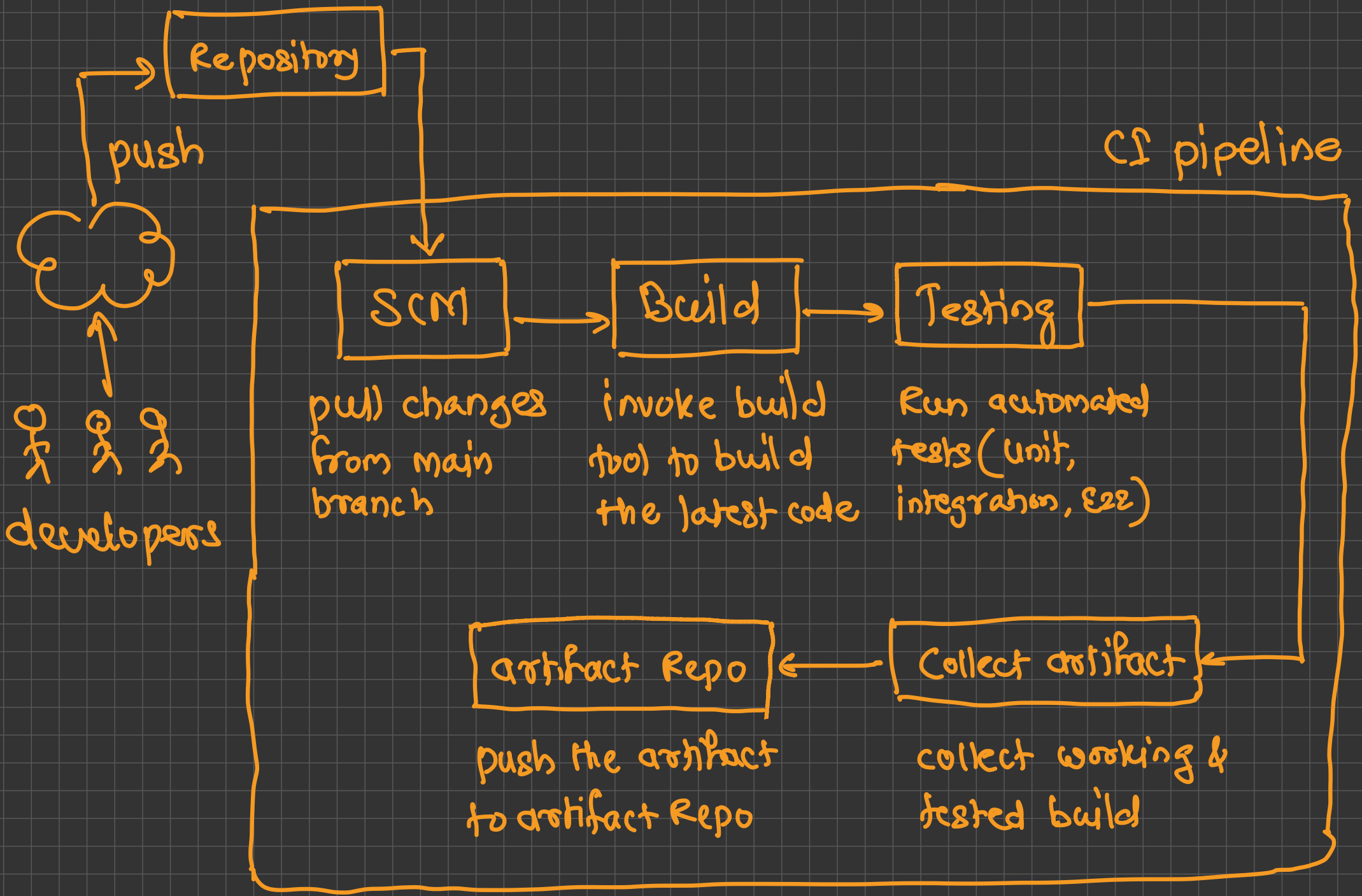
- working code
- well tested code
- non breaking code
- latest code having latest features

Core Principles of CI

- **Single Source Repository** → shared server → GitHub, Gitlab, BitBucket ....
  - All code lives in one version control system (Git, SVN, etc.). Everyone commits to the same repository, typically using a branching strategy like GitFlow or trunk-based development. → git → feature branch → svn/cvs
- **Frequent Commits**
  - Developers commit code at least once per day, ideally multiple times. Smaller, incremental changes are easier to integrate and debug than large batches.
- **Automated Build** → on main branch
  - Every commit triggers an automated build process that compiles the code and creates executable artifacts. This ensures the code can actually be built successfully. → build → package
- **Self-Testing Build** → automated testing build
  - The build includes running automated tests. If tests fail, the build is considered broken. Common test types include:
  - **Unit tests** - Test individual functions/components
  - **Integration tests** - Test how components work together → modules
  - **Static code analysis** - Check code quality and standards → static code analyzer → sonarqube → code coverage
  - **Security scans** - Identify vulnerabilities → VAPT
- **Fast Builds** → gradle, maven, xcodebuild
  - Builds should complete quickly (ideally under 10 minutes) so developers get rapid feedback. Slow builds discourage frequent commits.
- **Test in Production-Like Environment** → pre-production
  - CI environments should mirror production as closely as possible to catch environment-specific issues early.
- **Easy Access to Latest Build** → artifacts → artifact Repository → JFrog
  - Everyone on the team can access the latest executable and test results. Transparency is crucial.
- **Visible Build Status**
  - The current build status is highly visible to the team through dashboards, notifications, or physical indicators (like lava lamps or traffic lights in some offices).
- **Fix Broken Builds Immediately**
  - When a build breaks, fixing it becomes the top priority. The team shouldn't commit new changes until the build is green again.

The CI Workflow

- **Step 1: Developer Works Locally**
  - Developer pulls the latest code from the main branch
  - Makes changes and writes tests
  - Runs tests locally to ensure everything works
- **Step 2: Commit and Push**
  - Developer commits changes with a descriptive message
  - Pushes code to the central repository
- **Step 3: CI Server Detects Change**
  - The CI server (Jenkins, GitLab CI, etc.) monitors the repository
  - Detects the new commit via webhooks or polling
- **Step 4: Checkout Code**
  - CI server pulls the latest code from the repository
  - Creates a clean build environment
- **Step 5: Build Process**
  - Compiles the source code
  - Resolves dependencies
  - Creates build artifacts (executables, containers, packages)
- **Step 6: Run Tests**
  - Executes automated test suites
  - Generates test reports and code coverage metrics
  - May run multiple test stages in parallel for speed
- **Step 7: Code Quality Checks**
  - Runs linters to check coding standards
  - Performs static code analysis
  - Checks for security vulnerabilities
  - Verifies code coverage meets thresholds
- **Step 8: Generate Artifacts**
  - If all checks pass, creates deployable artifacts
  - Tags the build with version numbers
  - Stores artifacts in a repository (Artifactory, Nexus, container registry)
- **Step 9: Notification**
  - Sends results to the team (email, Slack, dashboard)
  - Updates build status badges
  - Green = success, Red = failure
- **Step 10: Feedback Loop**
  - If **successful**: Developer continues with next task
  - If **failed**: Developer investigates logs, fixes issues, and recommits


Repository
push
developers
CI pipeline
SCM
Build
Testing
pull changes from main branch
invoke build tool to build the latest code
Run automated tests (unit, integration, E2E)
artifact Repo
Collect artifact
push the artifact to artifact Repo
collect working & tested build

Key CI Components

- **Version Control System (VCS)**
  - Git (GitHub, GitLab, Bitbucket)
  - Subversion
  - Mercurial
- **CI Server/Platform**
  - **Jenkins** - Most popular, highly customizable, open-source
  - **GitLab CI/CD** - Integrated with GitLab, uses YAML configs
  - **GitHub Actions** - Native to GitHub, marketplace of actions
  - **CircleCI** - Cloud-based, fast Docker support
  - **Travis CI** - Popular for open-source projects
  - **TeamCity** - By JetBrains, powerful and user-friendly
  - **Azure DevOps** - Microsoft's comprehensive platform
  - **Bamboo** - By Atlassian, integrates with Jira
- **Build Tools**
  - **Maven, Gradle** (Java)
  - **npm, Yarn, Webpack** (JavaScript)
  - **MSBuild** (.NET)
  - **Make, CMake** (C/C++)
  - **pip, Poetry** (Python)

- **Build Tools**
  - **Maven, Gradle** (Java)
  - **npm, Yarn, Webpack** (JavaScript)
  - **MSBuild** (.NET)
  - **Make, CMake** (C/C++)
  - **pip, Poetry** (Python)

- **Testing Frameworks**
  - JUnit, TestNG (Java)
  - Jest, Mocha, Cypress (JavaScript)
  - pytest, unittest (Python)
  - NUnit, xUnit (.NET)
- **Code Quality Tools**
  - SonarQube - Comprehensive code analysis
  - ESLint, Pylint - Linters for specific languages
  - CodeClimate, Codacy - Cloud-based analysis
  - OWASP Dependency Check - Security scanning

Benefits of CI

- **Reduced Integration Risk** - Small, frequent merges prevent "integration hell"
- **Higher Code Quality** - Automated testing catches bugs early
- **Faster Feedback** - Developers know within minutes if their code works
- **Improved Collaboration** - Visibility into what everyone is working on
- **Reduced Manual Testing** - Automation frees up time for exploratory testing
- **Documentation** - Build logs provide a history of changes and issues
- **Confidence** - Teams can refactor and improve code without fear
- **Faster Time to Market** - Streamlined process accelerates delivery

Continuous Delivery and Continuous Deployment

- CD refers to two related but distinct practices that extend Continuous Integration:
- **Continuous Delivery** - Code changes are automatically built, tested, and prepared for release to production, but deployment requires manual approval. → manual deployment → configured for production env
- **Continuous Deployment** - Every change that passes all automated tests is automatically released to production without human intervention. → automatic deployment → all internal environments → any env
- Both ensure that code is always in a deployable state, but they differ in the final step to production.

The Problem CD Solves

- **Before CD**
  - Releases happened infrequently (monthly, quarterly, or even yearly)
  - Manual deployment processes were error-prone and stressful
  - Large releases contained many changes, making issues hard to trace
  - Deployment knowledge was siloed with specific team members
  - Testing in production-like environments happened too late
  - Rollbacks were complex and risky
  - Long lead times from code completion to customer value
- **With CD** → automation
  - Releases can happen multiple times per day
  - Deployments are automated, consistent, and repeatable
  - Small, incremental changes reduce risk
  - Anyone on the team can deploy with a button click
  - Issues are detected early through extensive automated testing
  - Rollbacks are quick and automated
  - Features reach customers rapidly

Continuous Delivery vs Continuous Deployment

- **Continuous Delivery**
  - **Manual gate** before production deployment
  - Humans decide **when** to release
  - Business decides the release schedule
  - Suitable when releases need coordination (marketing, training, etc.)
  - Common in enterprise, regulated industries, or when deploying to client-managed environments
  - **Example Flow:** Code → Build → Test → Stage → **[Manual Approval]** → Production
- **Continuous Deployment**
  - **Fully automated** pipeline to production
  - Every successful build goes live automatically
  - No human intervention in the deployment process
  - Requires mature testing and monitoring
  - Common in SaaS products, web applications, and companies like Netflix, Amazon, Etsy
  - **Example Flow:** Code → Build → Test → Stage → Production (all automated)

Core Principles of CD

- **Build Quality In** → Code quality
  - Quality isn't checked at the end; it's built into every step. Automated testing at multiple levels ensures confidence in releases.
- **Work in Small Batches**
  - Deploy small, incremental changes frequently rather than large batches. Smaller changes are easier to test, deploy, and troubleshoot.
- **Automate Everything** → CD tools
  - From building to testing to deployment, minimize manual steps. Automation ensures consistency and reduces human error.
- **Relentless Pursuit of Continuous Improvement**
  - Continuously optimize the pipeline, reduce cycle time, and improve feedback loops.
- **Everyone is Responsible**
  - The entire team owns the deployment pipeline, not just operations. Developers are accountable for production readiness.
- **Done Means Released**
  - Work isn't complete when code is written—it's complete when it's delivering value in production.

The CD Pipeline Stages

- **Stage 1: Source Control**
  - Everything starts with code committed to version control. This triggers the entire pipeline.
- **Stage 2: Build Automation**
  - Compile source code
  - Resolve and download dependencies
  - Create build artifacts (JAR files, Docker images, executables)
  - Version and tag artifacts
  - Store in artifact repository
  - **Tools:** Maven, Gradle, npm, Docker, Artifactory, Nexus
- **Stage 3: Automated Testing**
  - This is the most critical part of CD. Multiple layers of testing provide confidence:
  - Unit Tests
  - Integration Tests
  - Contract Tests → Interface testing
  - End-to-End Tests
  - Performance Tests
  - Security Tests

The CD Pipeline Stages

- **Stage 4: Staging/Pre-Production Deployment**
  - Code is deployed to an environment that mirrors production:
    - Same infrastructure configuration
    - Similar data volumes (often anonymized production data)
    - Same software versions
    - Equivalent network topology
  - **Activities:**
    - Deploy using the same process as production
    - Run full test suites
    - Perform exploratory testing
    - Validate configuration
    - Check monitoring and logging
- **Stage 5: Production Deployment**
  - The final deployment to live environment where users access the application
  - **Deployment Strategies:**
    - **Blue-Green Deployment**
    - **Canary Deployment**
    - Rolling Deployment

make sure app is working

→ pre-prod env

dummy

as prod env

– E2E

Infrastruction Creation

→ terrafoom, CF

→ automate infra building

↳ IaC tools

Benefits of CD

- **Faster Time to Market** - Features reach customers quickly
- **Reduced Risk** - Small, frequent deployments are less risky
- **Higher Quality** - Extensive automated testing catches issues
- **Lower Stress** - Deployments become routine, not events
- **Better Feedback** - Quick feedback from real users
- **Improved Developer Productivity** - Less time spent on manual processes
- **Competitive Advantage** - Respond to market changes rapidly
- **Lower Costs** - Automation reduces manual effort
- **Better Security** - Patches and fixes deployed quickly
- **Improved Reliability** - Consistent, tested deployment processes

Key CD Technologies and Tools

- Infrastructure as Code (IaC) tools
  - Define infrastructure in version-controlled files
  - Terraform - Multi-cloud infrastructure provisioning
  - CloudFormation - AWS-specific IaC
  - Ansible - Configuration management and provisioning
  - Chef, Puppet - Configuration management
  - Pulumi - IaC using programming languages

- Container Orchestration
  - Manage containerized applications
  - Kubernetes - Industry-standard orchestration
  - Docker Swarm - Simpler alternative to K8s
  - Amazon ECS/EKS - AWS container services
  - Azure AKS - Azure Kubernetes Service
  - Google GKE - Google Kubernetes Engine

- Deployment Tools
  - Automate the deployment process
  - Helm - Kubernetes package manager
  - ArgoCD - GitOps continuous delivery for K8s
  - Flux - GitOps toolkit for K8s
  - Spinnaker - Multi-cloud continuous delivery
  - Octopus Deploy - Deployment automation
  - AWS CodeDeploy - AWS deployment service

- Configuration Management
  - Manage application configuration
  - Consul - Service mesh and configuration
  - etcd - Distributed key-value store
  - Spring Cloud Config - Centralized configuration for Spring
  - AWS Systems Manager - Parameter Store and secrets
  - Azure Key Vault - Secrets management

Key CD Technologies and Tools

- Artifact Repositories
  - Store build artifacts
  - Docker Hub - Container images (public)
  - Amazon ECR - AWS container registry
  - Google Container Registry - GCP container registry
  - Artifactory - Universal artifact repository
  - Nexus - Repository manager
  - GitHub Packages - Integrated with GitHub

- Monitoring and Observability
  - Track application health:
  - Prometheus + Grafana - Metrics and visualization
  - Datadog - Full-stack monitoring
  - New Relic - APM and observability
  - Dynatrace - AI-powered monitoring
  - ELK Stack (Elasticsearch, Logstash, Kibana) - Log management
  - Splunk - Log analysis and SIEM
  - Jaeger, Zipkin - Distributed tracing
  - Sentry - Error tracking

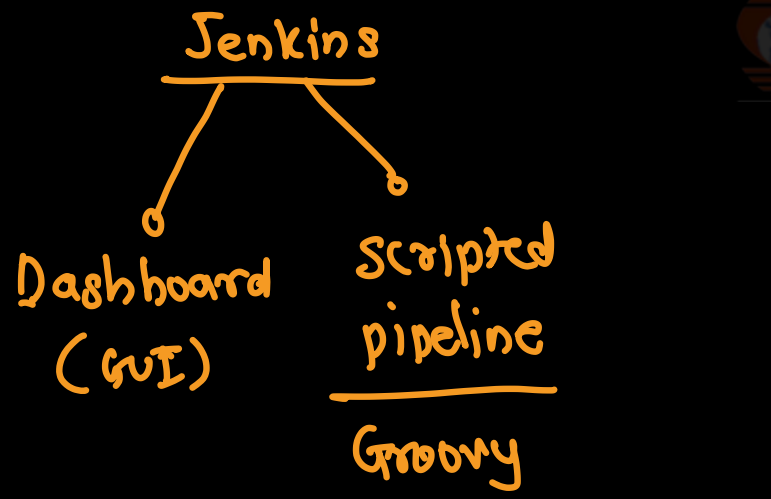
Jenkins

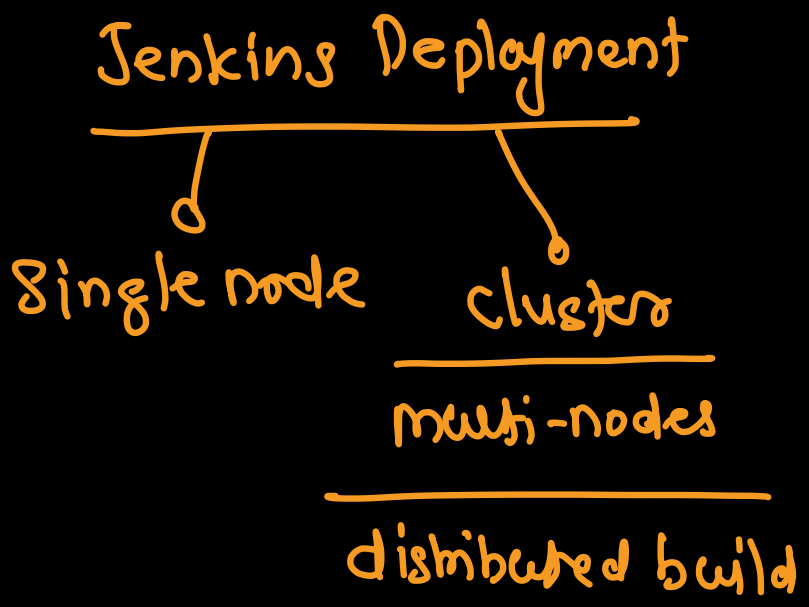
What is Jenkins ?

- **Jenkins** is an open-source automation server that enables developers to build, test, and deploy software reliably and efficiently , repeatedly
- It's the most widely used CI/CD tool in the world, with a massive ecosystem of plugins that extend its functionality
- Originally created as "Hudson" by Kohsuke Kawaguchi at Sun Microsystems in 2004, it was renamed Jenkins in 2011 after a community fork. Today, it's maintained by the Jenkins community and the Continuous Delivery Foundation

Why Jenkins

- **Strengths**
  - **Free and Open Source** - No licensing costs
  - **Highly Extensible** - Over 1,800 plugins available
  - **Platform Agnostic** - Runs on Windows, Linux, macOS
  - **Technology Agnostic** - Works with any programming language or tool
  - **Large Community** - Extensive documentation and community support
  - **Self-Hosted** - Full control over your CI/CD infrastructure
  - **Mature and Battle-Tested** - Used by thousands of organizations
  - **Flexible** - Can be configured for simple to extremely complex workflows
- **Weaknesses**
  - **Steep Learning Curve** - Complex to set up and configure initially
  - **Maintenance Overhead** - Requires dedicated resources for upkeep
  - **UI/UX** - Interface feels dated compared to modern alternatives
  - **Plugin Management** - Plugin compatibility issues can arise
  - **Scaling Challenges** - Requires careful architecture for large deployments
  - **Security** - Requires vigilant security configuration and updates



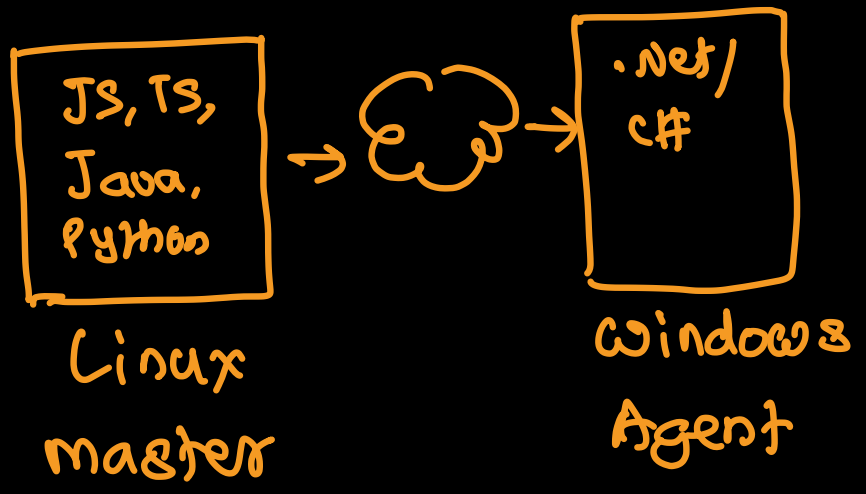


Architecture

Master - Agent (Slave)

- **Jenkins Master (Controller)**
  - The main Jenkins server
  - Manages the web interface → Dashboard
  - Schedules build jobs
  - Monitors agents
  - Distributes builds to agents
  - Records and presents build results
  - Stores configuration and build history
- **Jenkins Agents (Nodes/Slaves)**
  - Machines that execute build jobs
  - Receive instructions from the master
  - Execute build steps
  - Report results back to master
  - Can be static (always available) or dynamic (provisioned on-demand)
  - Can run on physical machines, VMs, containers, or cloud instances
  - **Executors**
    - Individual build slots on master or agents:
    - Each executor can run one build at a time
    - Master can have multiple executors (though not recommended for production)
    - Agents typically have multiple executors based on CPU cores

Distributed build

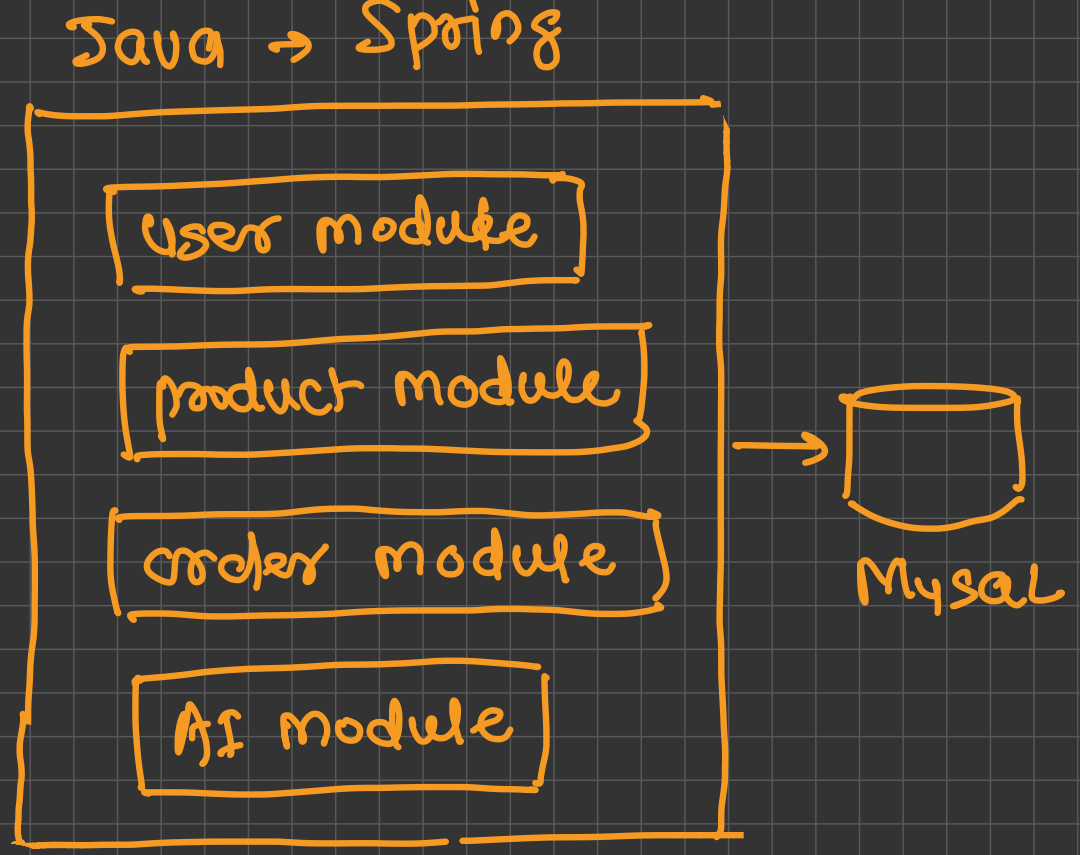


Monolithic architecture → one language
one server
one technology
one database
one repository

e.commerce

→ user module
→ product module
→ orders module
→ AI module
→ Notifications module

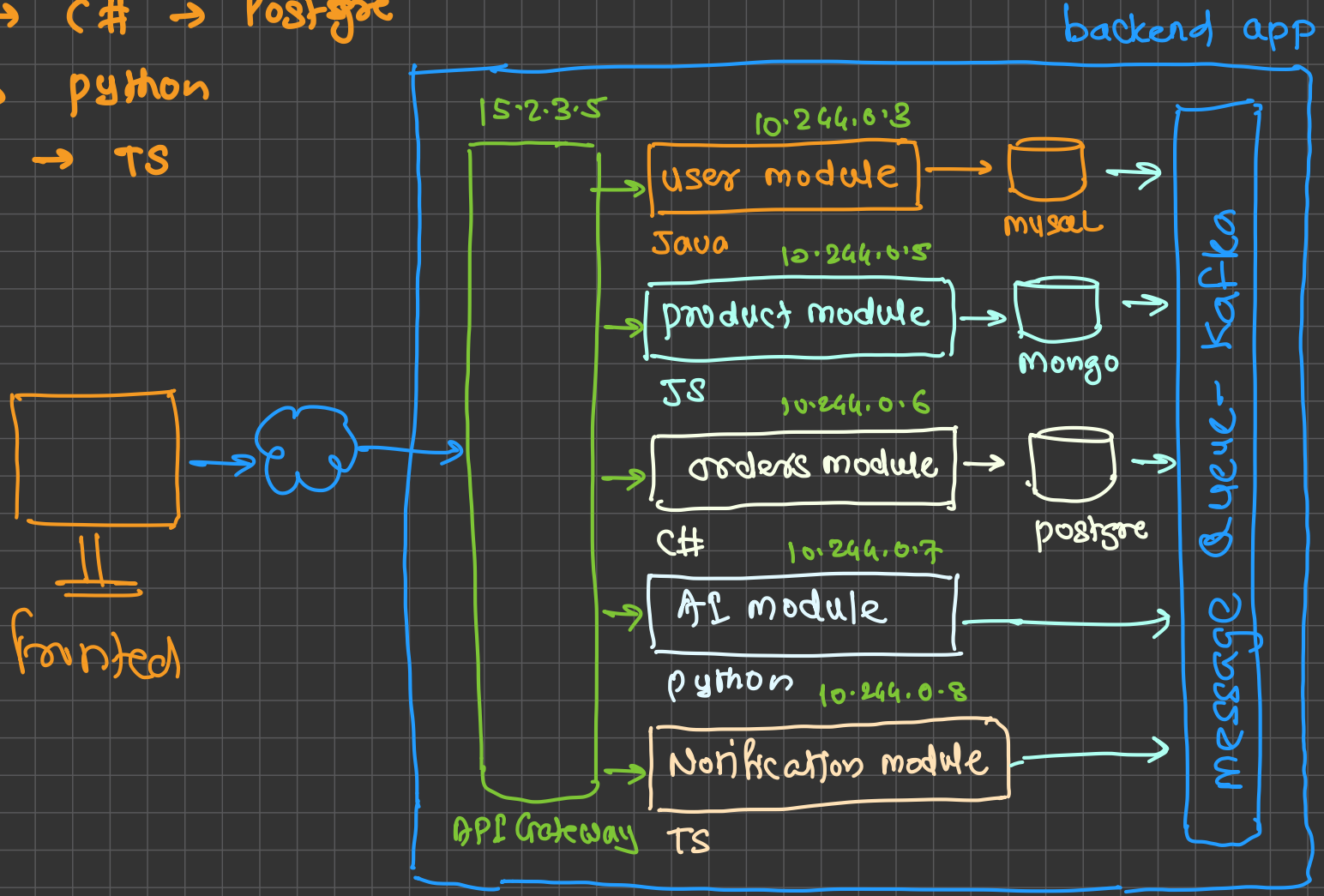
issues
- No language flexibility
- No DB flexibility
- Difficult scaling a part of app
- No fault isolation



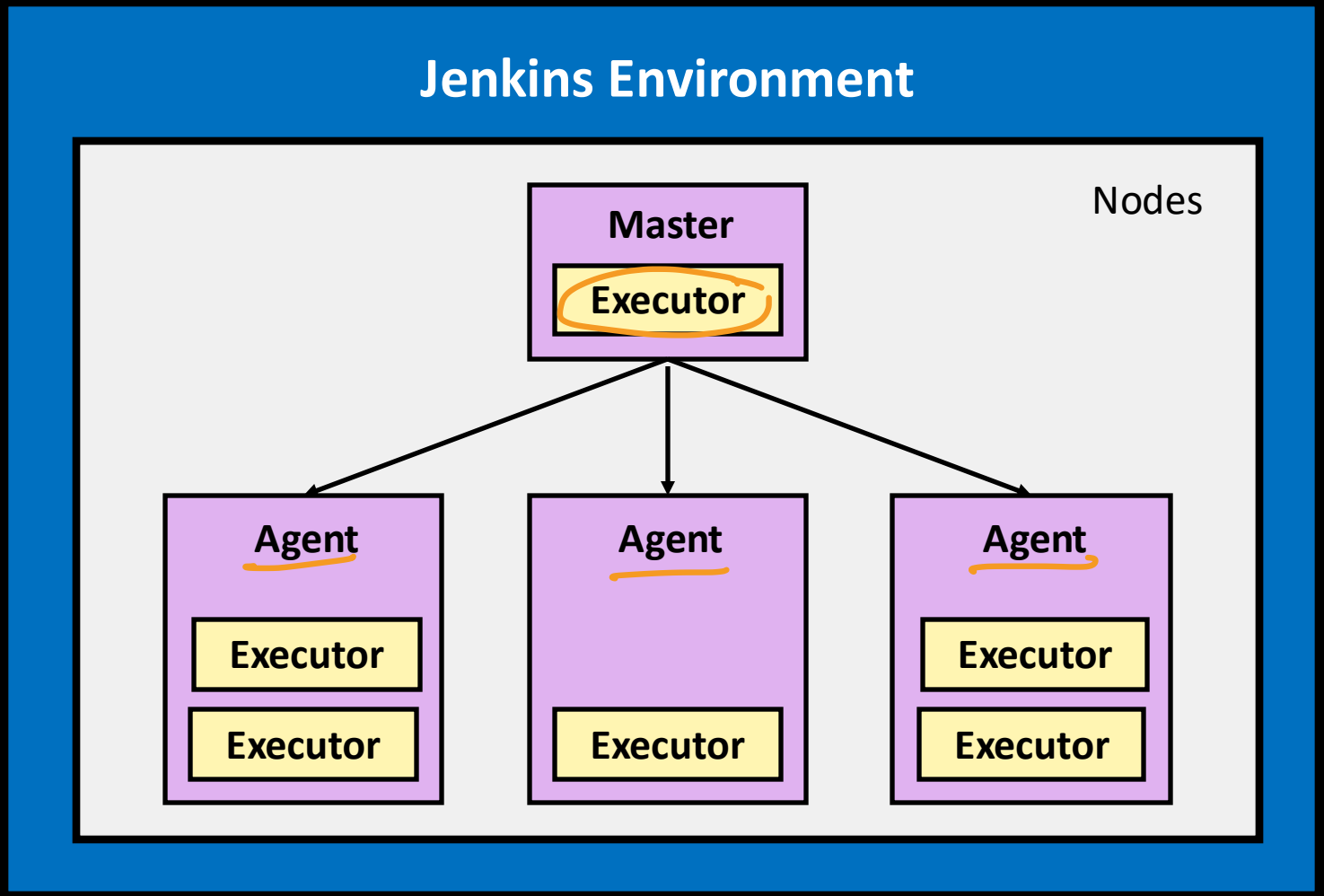
Micro-service architecture

e.commerce

- → user module → Java → MySQL
- → product module → JS → MongoDB
- → orders module → C# → Postgre
- → AI module → python
- → Notifications module → TS



Jenkins Environment



Installation

- Ubuntu/Debian
  - wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
  - sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
  - sudo apt update
  - sudo apt install jenkins
- Start Jenkins
  - sudo systemctl start jenkins
  - sudo systemctl enable jenkins