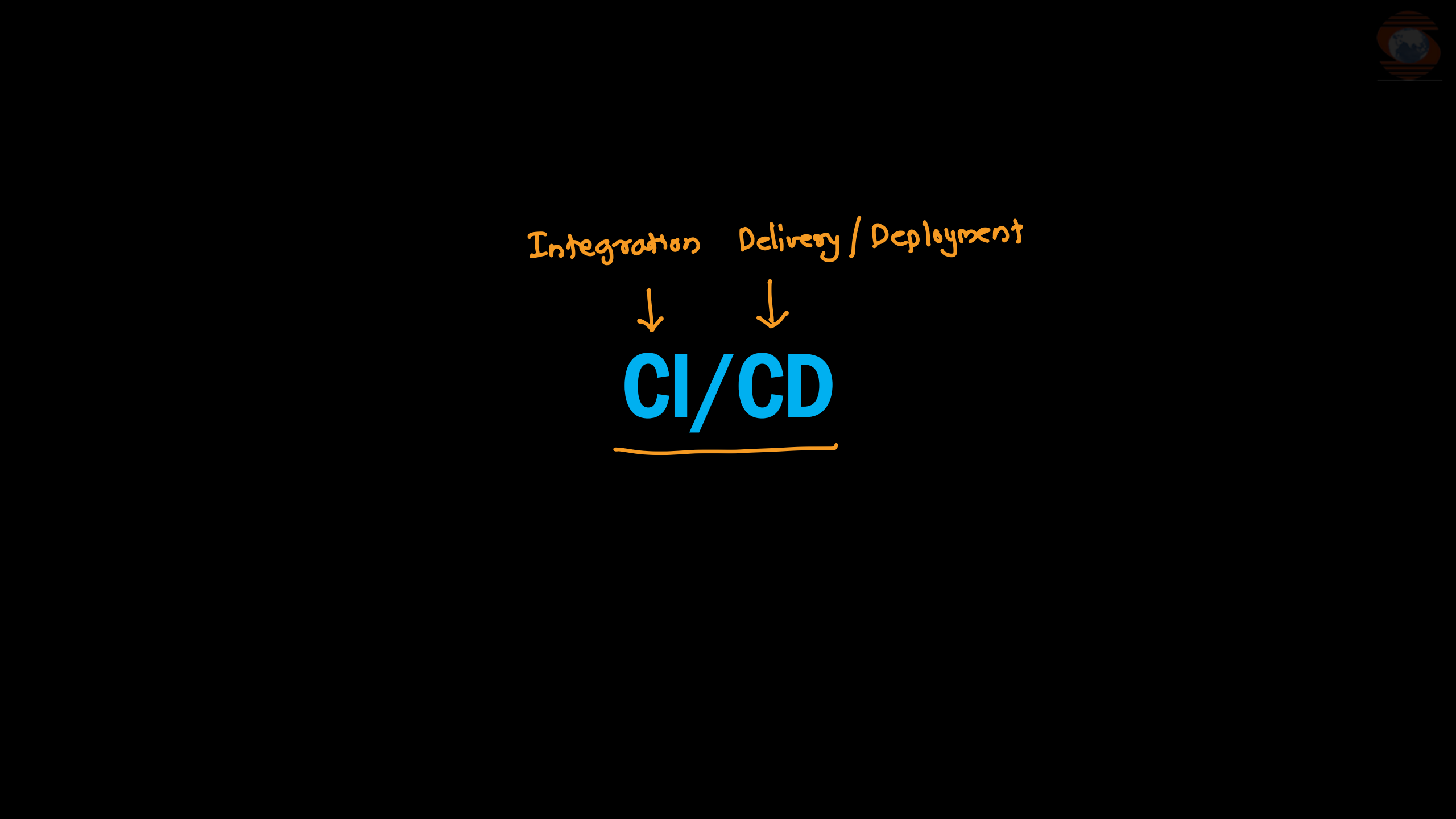
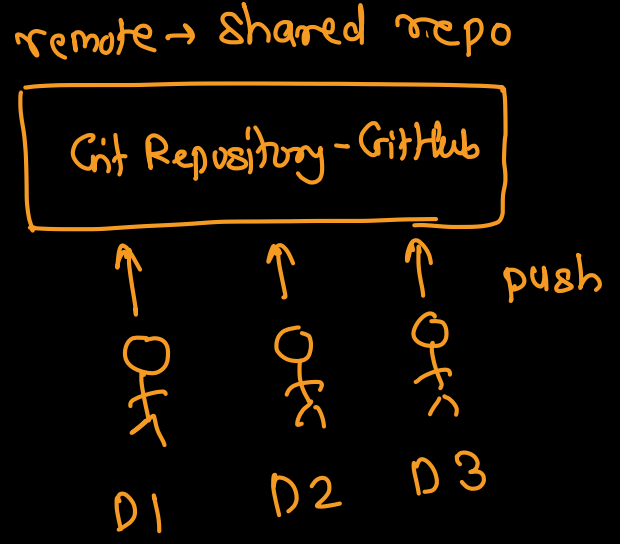

Integration
Delivery / Deployment
CI/CD

What is CI/CD Pinepines → Sequence of stages → scm, build, test, config, deploy

- A **CI/CD pipeline** automates the steps required from writing code - delivering the application - deploying it to production
- It ensures:
  - Faster development cycles
  - Higher code quality
  - Fewer manual errors
  - Repeatable and reliable deployments
- Benefits of CI/CD pipeline
  - **Business Benefits**
    - Faster time to market
    - Less downtime
    - Lower cost of development
    - Increased reliability → automation
  - **Developer Benefits**
    - Stable codebase
    - Automated workflows
    - Faster feedback loop
    - Easier collaboration

Continuous Integration

- 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
  - Teams get immediate feedback if something is wrong



The Problem CI Solves

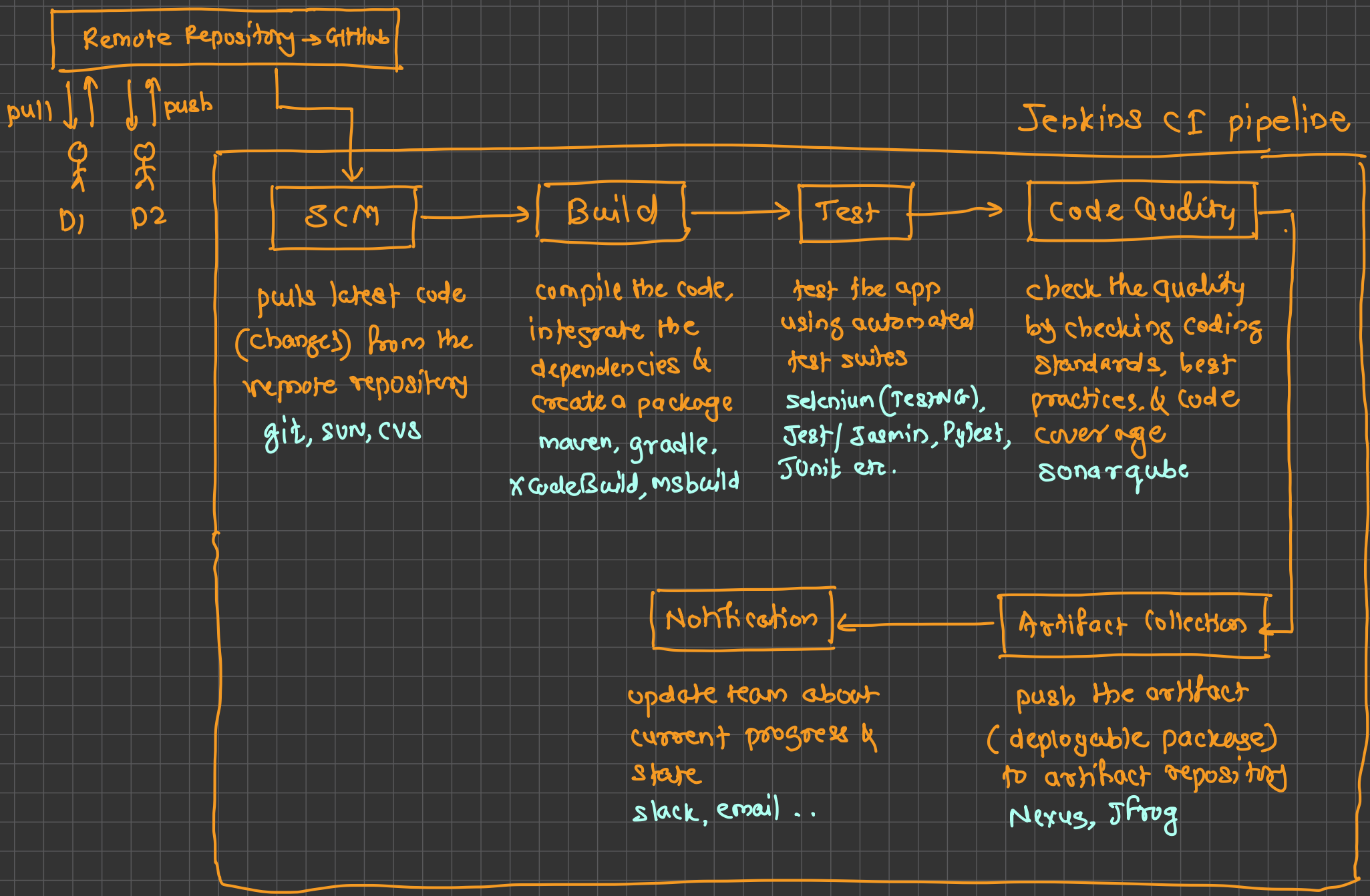
- **Before CI**
  - Developers worked on separate branches for weeks or months
  - When merging, massive conflicts occurred ("integration hell")
  - Bugs were discovered late, making them expensive to fix
  - No one knew if the combined code actually worked until the end
  - Building and testing were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

Core Principles of CI

- **Single Source Repository** (Remote)
  - 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.
- **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**
  - Every commit triggers an automated build process that compiles the code and creates executable artifacts (→ deployable package). This ensures the code can actually be built successfully.
- **Self-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
  - **Static code analysis** - Check code quality and standards
  - **Security scans** - Identify vulnerabilities
- **Fast Builds**
  - Builds should complete quickly (ideally under 10 minutes) so developers get rapid feedback. Slow builds discourage frequent commits.
- **Test in Production-Like Environment** → pre-prod / UAT
  - CI environments should mirror production as closely as possible to catch environment-specific issues early.
- **Easy Access to Latest Build**
  - Everyone on the team can access the latest executable and test results. Transparency is crucial.
- **Visible Build Status** → Dashboard
  - 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 → TDD
  - 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 → Sonarqube
  - Checks for security vulnerabilities → code coverage
  - 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


Remote Repository → GitHub
pull
push
D1
D2
Jenkins CI pipeline
SCM
Build
Test
Code Quality
pulls latest code (changes) from the repote repository
git, SVN, CVS
compile the code, integrate the dependencies & create a package
maven, gradle, XCodeBuild, MSbuild
test the app using automated test suites
selenium (TestNG), Jest/ Jasmin, Pytest, JUnit etc.
check the quality by checking coding standards, best practices & code coverage
sonarqube
Notification
Artifact Collection
update team about current progress & state
slack, email ..
push the artifact (deployable package) to artifact repository
Nexus, JFrog

Key CI Components

- **Version Control System (VCS)**
  - Git (GitHub, GitLab, Bitbucket)
  - Subversion - SVN
  - 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

manual deployment — automated 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
- **Continuous Deployment** - Every change that passes all automated tests is automatically released to production without human intervention. → automated deployment
- 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**
  - 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

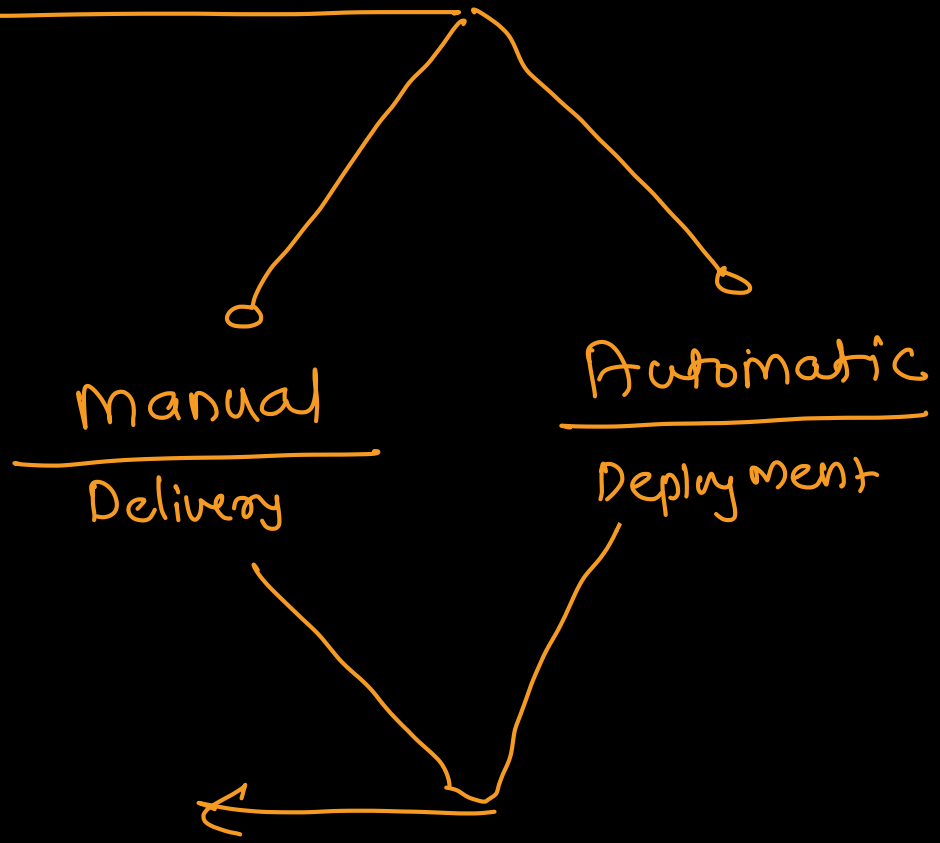
- **Build Quality In**
  - Quality isn't checked at the end; it's built into every step. Automated testing at multiple levels ensures confidence in releases.
- **Work in Small Batches**
  - Deploy small, incremental changes frequently rather than large batches. Smaller changes are easier to test, deploy, and troubleshoot.
- **Automate Everything**
  - 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**
  - **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



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)
  - 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

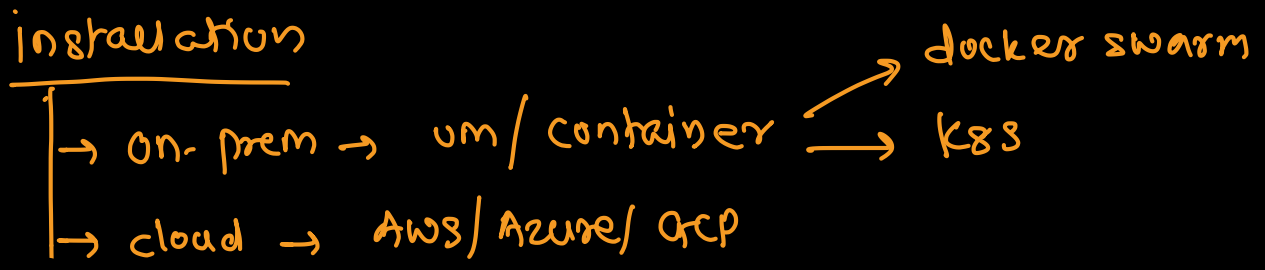
- Monitoring and Observability
  - Track application health:
  - Prometheus + Grafana - Metrics and visualization
  - Datadog - Full-stack monitoring
  - New Relic - APM and observability
  - Dynatrace - AI-powered monitoring
  - ELK Stack (Elasticsearch, Logstash, Kibana) - Log management
  - Splunk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

Jenkins

What is Jenkins ? → CI/CD tool

free

- Jenkins is an open-source automation server that helps software development teams automate the process of building, testing, and deploying their applications
- Think of Jenkins as a reliable assistant that can perform repetitive tasks automatically, saving developers time and reducing human errors
- It was originally called Hudson and was created in 2004 by Kohsuke Kawaguchi while he worked at Sun Microsystems
- In 2011, it was renamed to Jenkins after a dispute with Oracle, and it has grown to become the most popular automation server in the world
- Jenkins is free to use and has a large community of contributors who continuously improve it and create extensions for it
  - → plugins



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**

  ↳ scripted pipeline → Pipeline as a Code → Groovy
  - **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

monolithic architecture

→ single app has all the features

→ use one language, one technology, one repository & one database

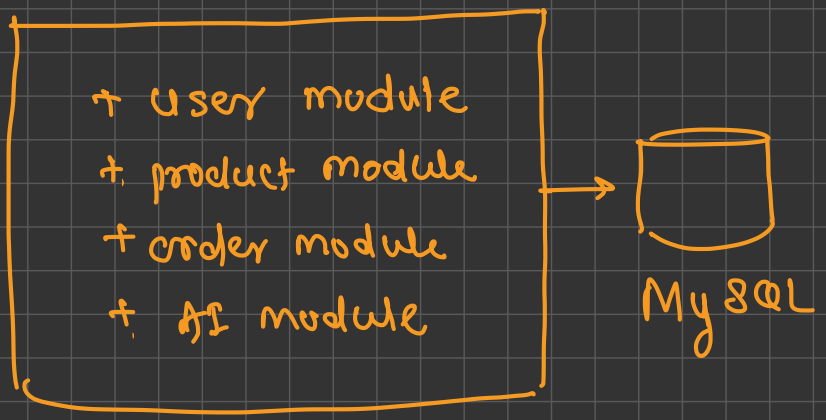
- → less learning curv
- → deployment is easy
- → faster than MS arch
- → less skills are required to maintain

- → No fault isolation
- → No flexibility for language & Database

→ e. commerce

- → user module
- → product module
- → order module
- → AI module

Repository



language → Java
framework → Spring

micro-services architecture

→ instead of one big app, MS is used to implement single functionality

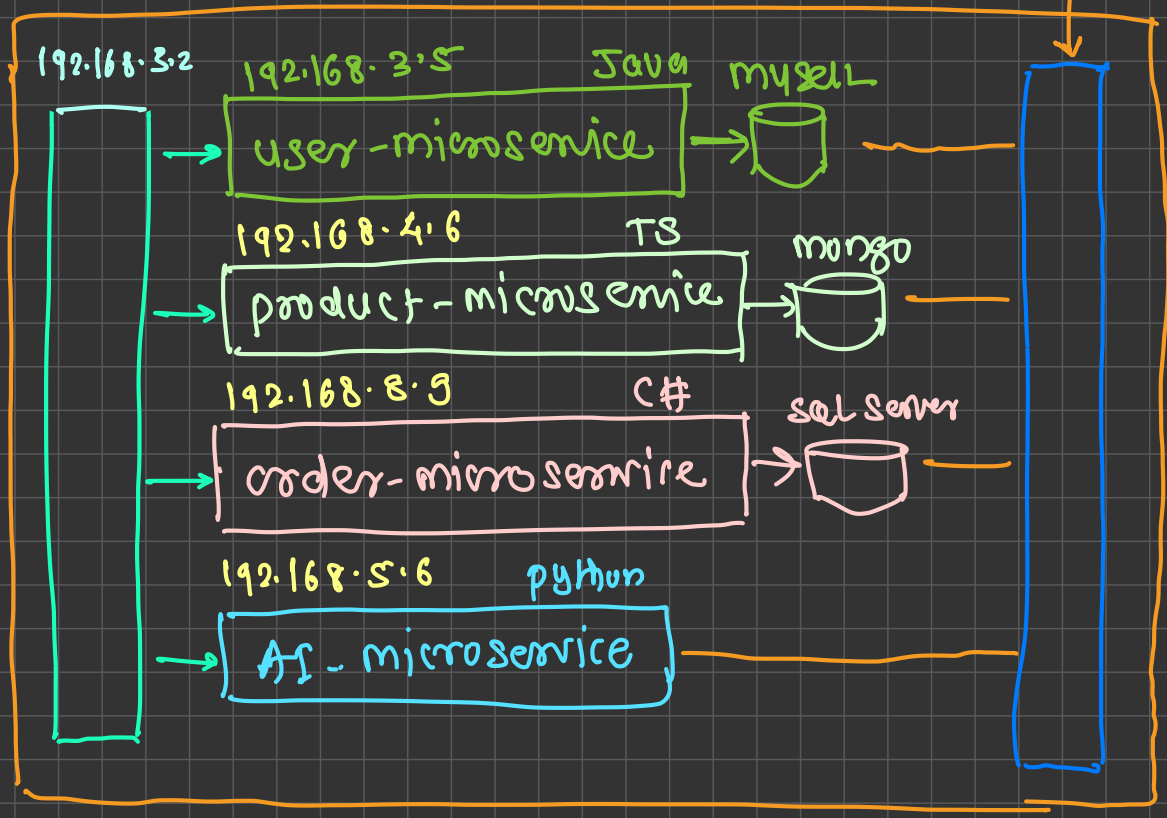
→ collection of multiple smaller applications called as MS

- → flexibility on language & DB
- → fault isolation
- → easy scaling

- → complex architecture → slower & costly
- → overhead
- → more skills are required
- → more steep learning curv

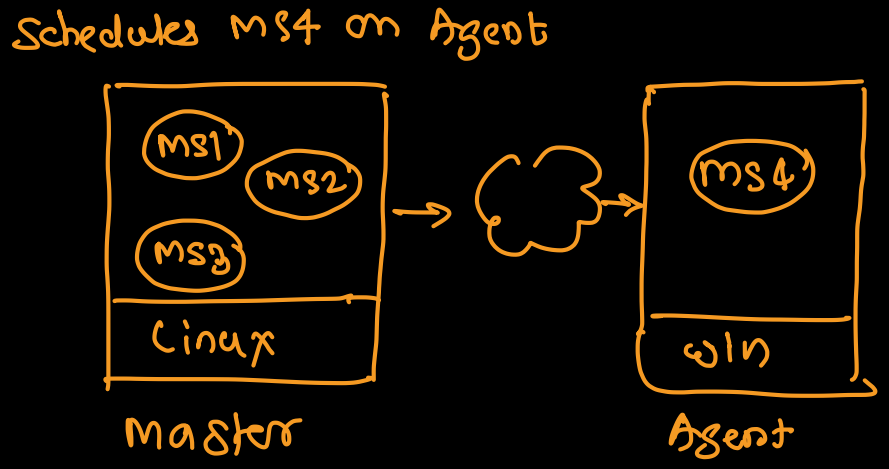
→ e-commerce

- → user module → Java / MySQL
- → product module → TS / Mongo
- → order module → C# / MS SQL server
- → AI module → python



Architecture

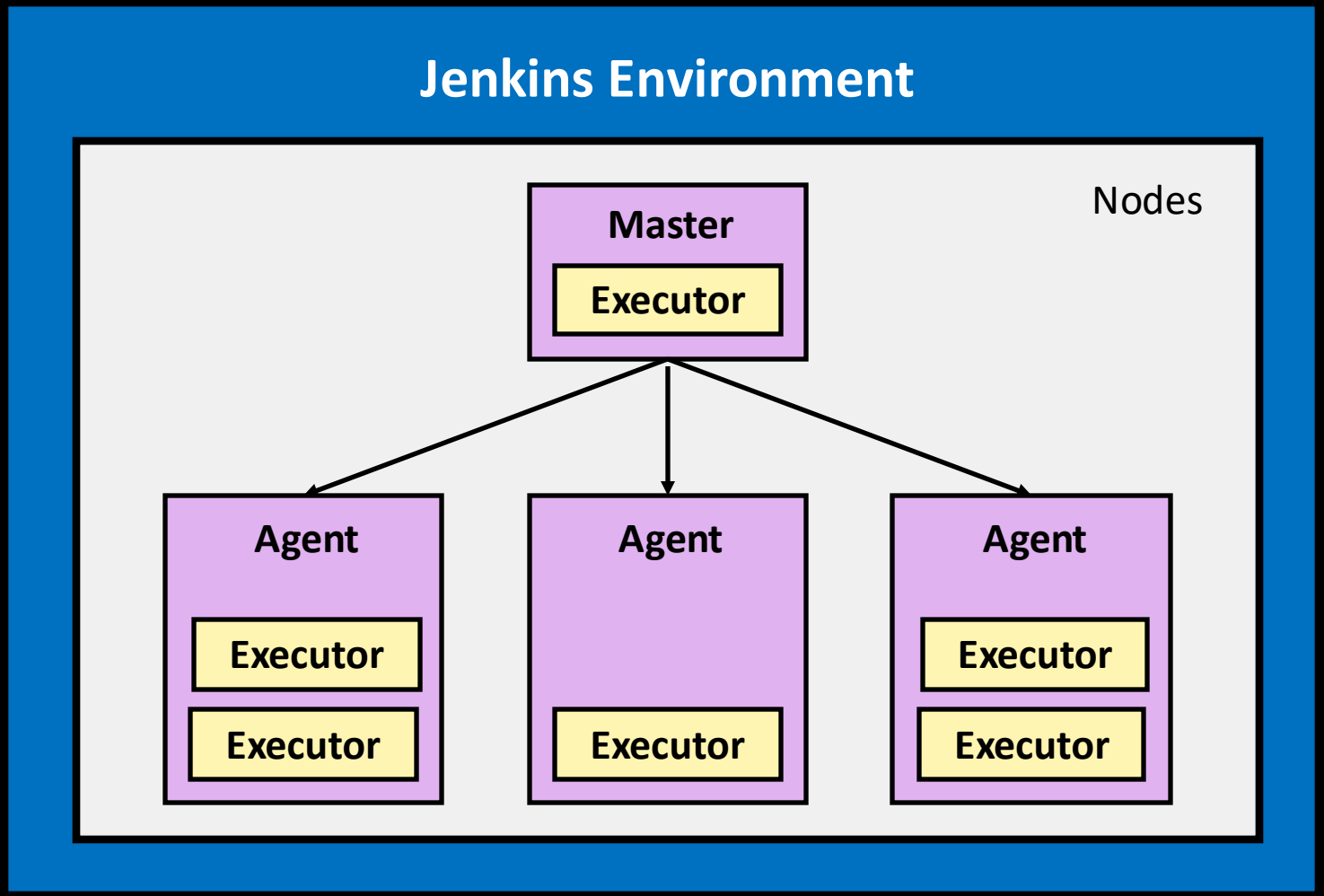
- Jenkins operates using a master-agent architecture, which means it has one central controller (the master) and can have multiple workers (agents)
- The master server is the brain of Jenkins - it schedules builds, monitors agents, records results, and provides the web interface you interact with
- Agents are separate machines that actually execute the work - they compile code, run tests, and perform deployments as instructed by the master
- This architecture allows Jenkins to distribute work across multiple machines, enabling it to handle many projects simultaneously
- For small projects, you might run everything on the master, but for larger organizations, you would have dozens or hundreds of agents



Node Types

- Master Server
  - The master server is where Jenkins itself runs and where you configure all your automation tasks through a web browser
  - It stores all the configuration data, including information about what tasks to run, when to run them, and where to run them
  - The master continuously monitors your code repositories (like GitHub or GitLab) for changes and triggers builds when new code is committed
  - It also maintains the build history, showing you which builds succeeded, which failed, and providing access to logs and test results
  - Think of the master as the conductor of an orchestra, coordinating all the different pieces but not necessarily playing every instrument itself
- Agent Node
  - Agents (also called slaves or nodes) are separate computers or virtual machines that connect to the Jenkins master to perform actual work
  - When the master decides a build needs to run, it sends instructions to an available agent, which then executes those instructions
  - Agents can run on different operating systems (Windows, Linux, Mac) allowing you to test your application on multiple platforms simultaneously
  - Each agent can have labels describing its capabilities, like "linux," "docker," or "high-memory," helping Jenkins choose the right agent for each job
  - Using multiple agents allows your organization to run many builds in parallel, dramatically reducing wait times when multiple developers are committing code

Jenkins Environment



Jenkins Job

- Freestyle jobs are the most basic type - they provide a simple web form where you configure each step of your automation process
- Pipeline jobs use code (written in a language called Groovy) to define your automation, making it easier to manage complex workflows and version control your automation scripts
- Multi-configuration jobs allow you to run the same job with different parameters, like testing your application on Windows, Linux, and Mac simultaneously
- Folder jobs help organize your jobs into hierarchies, similar to how you organize files on your computer into folders
- Multi-branch pipeline jobs automatically create a separate pipeline for each branch in your source code repository, perfect for teams working on multiple features simultaneously

Jenkins Builds

- A build is a single execution of a Jenkins job - every time a job runs, it creates a new build with an incrementing number
- E.g., if you have a job called "Build My App," its first execution is Build 1, the second execution is Build 2, and so on
- Each build has a status: Success (everything worked), Failure (something went wrong), Unstable (it ran but tests failed), or Aborted (someone stopped it)
- Jenkins keeps a history of all builds, allowing you to see trends over time, like whether your project is getting more stable or less stable
- You can view logs for each build to see exactly what happened during execution, which is invaluable for troubleshooting when something goes wrong

Jenkins Workspace

- A workspace is a directory on an agent's hard drive where Jenkins stores the files needed for a particular job
- When a build starts, Jenkins typically checks out your source code from a repository into the workspace, giving the build access to your files
- The workspace persists between builds of the same job on the same agent, which can speed up subsequent builds by avoiding redundant downloads
- However, this also means that leftover files from previous builds might interfere with new builds, so Jenkins provides options to clean the workspace before each build
- Each job usually has its own workspace directory to prevent different jobs from interfering with each other's files

Jenkins Plugins

- Plugins are extensions that add new functionality to Jenkins, similar to how apps add features to your smartphone
- Jenkins ships with basic functionality, but plugins allow you to integrate with hundreds of different tools and services like GitHub, Docker, Kubernetes, and Slack
- For example, the Git plugin lets Jenkins check out code from Git repositories, while the Email plugin lets Jenkins send notification emails
- There are over 1,800 plugins available in the Jenkins plugin repository, covering everything from source control to deployment to notifications
- Plugins are maintained by the Jenkins community, and you can install them through Jenkins' web interface with just a few clicks

Installing Jenkins - System Requirements

- Before installing Jenkins, your computer or server needs Java installed because Jenkins is a Java application
- Specifically, Jenkins requires Java 11 or Java 17 (these are versions of the Java runtime environment)
- You will need at least 256 MB of RAM, but 1 GB or more is recommended for any real-world usage
- About 10 GB of hard drive space is recommended - 1 GB for the Jenkins application itself and therest for storing build data and workspaces
- Jenkins can run on Windows, Linux, or Mac operating systems, and there are installation packages available for all major platforms
- 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