# Module 1: Introduction to DevOps and Continuous Delivery

**Introduction to DevOps and Continuous Delivery**

- **What is DevOps?**
  DevOps is a software development methodology that combines development (Dev) and operations (Ops) teams into a unified approach. It aims to shorten the software development life cycle, improve collaboration between teams, and increase the quality and speed of software delivery. DevOps emphasizes automation, continuous integration, continuous delivery, and feedback loops. The core values include collaboration, transparency, and shared responsibility for the end product.

- **How Fast is Fast?**
  In DevOps, speed is a critical factor in delivering features to customers quickly. The main goal is to reduce the time between writing code and deploying it to production, allowing businesses to be more responsive to market changes. However, "fast" doesn't mean rushing. It emphasizes doing so while maintaining the quality and stability of the software.

- **The Agile Wheel of Wheels**
  DevOps draws from Agile principles, focusing on iterative development and feedback. The "wheel of wheels" refers to the continuous feedback loop where the product is built, tested, deployed, and improved in short cycles. This methodology allows for quick delivery and responsiveness to customer feedback.

- **Beware the Cargo Cult Agile Fallacy**
  Cargo cult Agile refers to organizations adopting Agile practices without fully understanding the underlying principles. It often leads to ineffective implementation, such as ceremonies being performed without the expected results. The fallacy arises when teams follow Agile rituals, such as daily standups or sprint planning, but fail to embrace its core values, like collaboration and adaptability.

**A View from Orbit**

- **DevOps Process and Continuous Delivery**
  DevOps is an end-to-end approach that encompasses development, testing, deployment, and operations. Continuous Delivery (CD) ensures that software is always in a deployable state, which allows for quick releases and faster feedback. The DevOps process integrates these steps, providing smooth transitions from development to production through automated pipelines.

- **Release Management**
  Release management involves planning, scheduling, and controlling software deployments. In DevOps, release management is automated to ensure continuous delivery of features and fixes to users without manual intervention. Proper release management ensures that each new version of the software can be tested, released, and deployed quickly and reliably.

- **Scrum and Kanban**
    - **Scrum**: Scrum is a framework that uses sprints, which are time-boxed iterations, to deliver value in increments. Scrum teams hold regular ceremonies such as sprint planning, daily standups, sprint reviews, and retrospectives.
    - **Kanban**: Kanban is a visual management method that focuses on managing and improving flow. It uses a board with columns to represent different stages of work and limits the number of tasks that can be worked on at a time.
- **Delivery Pipeline**
  A delivery pipeline is an automated process that allows software to go from development to production through stages like build, test, and deployment. Each stage is automated to ensure the software quality and readiness for production. This helps in minimizing human errors and reducing deployment time.
- **Wrapping Up – A Complete Example**
  A complete example would showcase a scenario where DevOps and continuous delivery practices are applied. This could include code being written, committed, automatically tested, deployed to a staging environment, and then pushed to production, all with minimal human intervention.
- **Identifying Bottlenecks**
  Bottlenecks are points where the flow of work is impeded. These can be in the build, test, deployment, or operations phases. Identifying and addressing these bottlenecks helps in speeding up the entire process. In a DevOps environment, tools like Jenkins, Prometheus, and Grafana can be used to monitor the pipeline and locate these inefficiencies.

---

# Module 2: Everything is Code

### The Need for Source Code Control

Source code control (version control) is essential for tracking changes in the codebase, enabling collaboration between developers, and maintaining a history of the code. It helps in keeping track of who made changes, why they were made, and allows developers to roll back to previous versions if necessary.

### The History of Source Code Management

Source code management has evolved from manual methods to more advanced tools. Early systems required developers to track changes manually, while modern systems like Git allow developers to track changes, manage branches, and collaborate more efficiently. The evolution has significantly improved the speed and reliability of software development.

**Roles and Code**

In software development, various roles contribute to managing and maintaining code. These include:

- **Developers**: Write the code and create features.
- **Testers**: Ensure the code works as expected.
- **DevOps Engineers**: Automate the deployment pipeline.
- **Product Managers**: Define the features to be built. Each of these roles interacts with source code, but each has a different perspective and responsibility.

**Which Source Code Management System?**

There are various source code management systems available:

- **Git**: The most popular modern version control system, known for its distributed nature.
- **SVN (Subversion)**: An older, centralized version control system.
- **Mercurial**: Another distributed version control system, similar to Git. Choosing the right system depends on team needs, with Git being the most widely adopted.

**A Word About Source Code Management System Migrations**

Migrating from one source code management system to another can be challenging, especially when dealing with large codebases. Teams must plan carefully to ensure that code history is preserved and that the transition does not disrupt the workflow.

**Choosing a Branching Strategy**

Branching strategies define how developers manage different versions of the code:

- **Feature Branching**: Each feature is developed in its own branch.
- **Git Flow**: Involves multiple long-lived branches such as main, develop, feature, and release.
- **Trunk-Based Development**: Developers work on the main branch continuously with frequent commits.

**Branching Problem Areas**

Branching can lead to issues such as merge conflicts, long-lived branches, and complex dependency chains. These challenges require good management and frequent synchronization to avoid integration problems.

**Artifact Version Naming**

When creating build artifacts (like JAR files or Docker images), consistent versioning is crucial for traceability and managing dependencies. It helps teams track which versions of code have been deployed and which versions need updating.

### Choosing a Client

Clients are tools used to interact with source code management systems. Popular Git clients include:

- **Git Command Line**: Provides full control over Git operations.
- **GitHub Desktop**: A GUI-based client for GitHub users.
- **SourceTree**: A free Git client by Atlassian.

### Setting Up a Basic Git Server

To set up a Git server, you can install software like **GitLab** or **Gitea**, or even host your own server using Git via SSH. This allows teams to manage and share code securely.

### Shared Authentication

To enhance security and streamline access to repositories, shared authentication methods like SSH keys or OAuth can be used. These methods ensure that only authorized users can access the repositories.

### Hosted Git Servers

Hosted Git servers like **GitHub**, **GitLab**, and **Bitbucket** provide cloud-based solutions for managing source code. These platforms offer additional features like issue tracking, continuous integration, and deployment pipelines.

---

## Module 3: Building the Code

### Why Do We Build Code?

The purpose of building code is to transform it from source code into a final, executable form. Building ensures that code compiles without errors and is ready to be run, tested, and deployed.

### The Many Faces of Build Systems

A build system automates the process of compiling code, running tests, packaging artifacts, and generating the final executable. Different tools like **Maven**, **Gradle**, and **Make** are used to define and manage build processes.

### The Jenkins Build Server

Jenkins is an open-source automation server used to implement CI/CD pipelines. It automates the process of building, testing, and deploying code. Jenkins supports various plugins to integrate with source control systems, testing frameworks, and deployment tools.

### Managing Build Dependencies

Build dependencies are external libraries or tools required for building the software. Tools like **Maven** (for Java) or **npm** (for JavaScript) are used to manage dependencies and ensure that the correct versions of libraries are used.

**The Final Artifact**

The final artifact is the output produced by the build process, such as compiled code, Docker images, or packaged files. These artifacts are then used in deployment to the production environment.

**Continuous Integration (CI)**

Continuous Integration involves frequently merging code changes into a central repository where automated tests are run to detect errors early. CI ensures that code changes are integrated smoothly and that the software remains in a deployable state.

**Continuous Delivery (CD)**

Continuous Delivery extends Continuous Integration by automating the deployment of the application to production. CD ensures that the software can be released at any time with minimal manual intervention.

**Jenkins Plugins**

Jenkins plugins add functionality to the Jenkins server, such as integration with Git, Docker, Kubernetes, Slack, and more. They enable Jenkins to perform tasks like running tests, sending notifications, and deploying code.

**The Host Server**

The host server is the infrastructure where Jenkins and other build tools are executed. This can be a physical machine, a virtual machine, or a cloud-based service.

**Build Slaves**

In Jenkins, build slaves are machines that perform the actual work of building and testing code. They can be distributed across different environments, improving scalability.

**Software on the Host**

The software installed on the host server must include compilers, build tools, and dependencies required to run the builds. For example, for a Java project, the Java Development Kit (JDK) would need to be installed on the host server.

**Triggers**

Build triggers in Jenkins can automatically start a build process based on events, such as code commits, scheduled intervals, or manual intervention.

**Job Chaining and Build Pipelines**

Job chaining refers to linking multiple Jenkins jobs in a sequence where the output of one job is used as the input for the next. Build pipelines automate the entire process from code integration to deployment.

### A Look at the Jenkins Filesystem Layout

Understanding the Jenkins filesystem helps administrators troubleshoot issues related to workspace and configuration files. Jenkins typically stores build artifacts and logs in dedicated directories.

### Build Servers and Infrastructure as Code

Infrastructure as Code (IaC) involves managing and provisioning infrastructure using code and automation tools like **Terraform** or **Ansible**. This ensures that the infrastructure is consistent, reproducible, and version-controlled.

### Build Phases

Build phases represent different stages in the build lifecycle, such as compilation, testing, packaging, and deployment. Each phase ensures that code is validated and ready for the next step.

### Alternative Build Servers

Alternative build servers like **GitLab CI/CD**, **CircleCI**, and **Travis CI** also provide automated build, test, and deployment pipelines with varying features and integrations.

### Collating Quality Measures

Quality measures involve assessing the code's health through automated tools like **SonarQube**, which analyze code for bugs, vulnerabilities, and code smells.

### About Build Status Visualization

Build status visualization helps track the health of the codebase through dashboards or notifications. This allows teams to quickly identify failed builds and prioritize fixes.

### Taking Build Errors Seriously

Build errors should be taken seriously as they may indicate underlying issues in the code. Promptly addressing build errors prevents delays in the development cycle and ensures the quality of the code.