Week 1

❖ What is Version Control

- Version control, also known as source control or source code management, is a system that records all changes and modifications to files for tracking purposes.
- It provides developers with the ability to access the entire change history and revert to previous states or points in time.

❖ Types of Changes

- Version control systems handle various types of changes, including adding new files, modifying or updating files, and deleting files.

❖ Benefits of Version Control

- Revision History: Version control systems maintain a record of all changes, allowing developers to revert to stable points if issues or bugs arise.
- Identity: Every change is recorded with the identity of the user who made it, enabling accountability and transparency.
- Collaboration: Teams can collaborate effectively by submitting and tracking code changes. Peer reviews facilitate code inspection and feedback.
- Automation: Version control is a cornerstone of DevOps, aiding in software quality, release management, and deployments.
- Efficiency: Version control enhances team efficiency, particularly in Agile development, by enabling effective task management and automation.

✓ Version control is an essential tool for software developers, as it promotes collaboration, accountability, and efficiency. Understanding and using version control systems are fundamental skills for modern software development.

❖ Types of Version Control Systems:

- There are various version control systems available, including Subversion, Perforce, AWS Code Commit, Mercurial, and Git.

Week 1

❖ Centralized Version Control Systems (CVCS):

- CVCS has a server-client architecture.
- The server holds the main repository with the complete history of code versions.
- Developers need to pull code from the server to have their working copy.
- All operations require a connection to the central server.
- Changes made by developers must be pushed to the central server to share with the team.
- Viewing the history of changes depends on being connected to the server.

❖ Distributed Version Control Systems (DVCS):

- DVCS is similar to CVCS but with a crucial difference.
- In DVCS, every user is essentially a server, not just a client.
- When you pull code in DVCS, you have the entire history of changes locally.
- You don't need a constant server connection to work, add changes, or view file history.
- Server connectivity is only required to pull or push changes.
- DVCS allows users to work offline effectively.

❖ Advantages and Disadvantages:

- CVCS is often considered easier to learn.
- CVCS provides more access controls to users.
- DVCS offers better speed and performance as most operations occur locally.
- DVCS enables offline work, making it more efficient and suitable for distributed teams.
- DVCS has a significant impact on improving developer operations and the software development life cycle.

❖ Concurrent Versions System (CVS):

Developed in the 1980s by Walter F. Tichy at Purdue University and released publicly in 1990, CVS was one of the earliest version control systems. It allowed developers to track changes to files and directories, making it easier to collaborate on software projects.

Week 1

❖ Subversion (SVN):

Developed by CollabNet in 2000, SVN addressed some of the limitations of CVS. It introduced integrity checks and improved support for versioning binary files. SVN became popular in the open-source community and offered free hosting for open-source projects.

❖ BitKeeper and the Birth of Distributed VCS:

BitKeeper, a proprietary distributed version control system, was used for Linux kernel development. However, in 2005, a licensing dispute led to the discontinuation of its free use. This event prompted the development of new distributed VCS systems.

❖ Mercurial:

Created by Matt Mackall, Mercurial was designed as a high-performance distributed VCS. It gained popularity, especially among former SVN users, due to its ease of use and smooth transition process. Many hosting platforms began offering Mercurial hosting.

❖ Git:

Developed by Linus Torvalds in response to the BitKeeper controversy, Git was designed to host the Linux kernel's source code. Git's distributed VCS model and its first public release in 2007 made it a powerful and flexible version control system. GitHub's offering of free Git hosting for open-source projects further accelerated its adoption.

Git, in particular, has become the de facto standard for version control in both open-source and proprietary software development. Its distributed nature, speed, and extensive ecosystem of tools and services have made it an integral part of modern software development workflows.

Week 1

❖ Workflow:

Having a well-defined workflow is crucial in managing code changes, especially in collaborative environments. It helps address conflicts, defines how code is reviewed, and ensures a systematic approach to development. Different projects may adopt different workflows, and in this course, you'll learn common Git-based workflows.

❖ Continuous Integration (CI):

CI is a practice where code changes are frequently integrated into a shared repository. Automated tests are run to validate these changes, ensuring that the codebase remains stable. CI helps catch issues early and reduces the likelihood of merge conflicts.

❖ Continuous Delivery (CD):

CD extends the concept of CI. Once code changes are integrated and tested, a CD system automatically packages the application, making it ready for deployment. This reduces the risk of human error when preparing the application for deployment.

❖ Continuous Deployment:

Going a step further, continuous deployment aims to automatically release software to customers after successful testing. It typically involves deploying to a test environment first to validate changes and then to the live production environment once they are confirmed to be stable. This approach allows for frequent and safe releases.

❖ Source of Truth:

Version control systems act as a "source of truth" for the codebase, maintaining a comprehensive history of all changes made to every file in the repository. This history serves as a valuable reference for developers and team members.

Week 1

❖ Collaboration:

Version control facilitates collaboration within development teams. It enables developers to work together on a common goal, whether it's adding new features or identifying and resolving potential issues.

❖ Revision History:

The revision history records essential information about each change, including who made it, when it was made, and what exactly was changed. This information is easily accessible, either through commands or integration with development environments (IDEs).

❖ Communication and Transparency:

Establishing communication standards for developers is crucial. Clear and detailed commit messages help team members understand the purpose and context of each change. This transparency fosters a more cohesive and open team environment.

❖ Development Environments:

Development teams typically set up multiple environments to test and verify code changes before releasing them to production. These environments serve different purposes and help ensure code quality and reliability.

❖ Staging Environment:

The staging environment should closely mimic the production environment. It's where teams test code changes to identify and resolve potential issues before deploying to production. Staging covers all aspects of the application architecture, including databases and other required services. It's beneficial for testing new features, running various types of tests (e.g., unit, integration, performance), and validating data migrations and configuration changes.

Week 1

❖ Production Environment:

This is the live environment where end-users interact with the application. Code deployed to production should have been thoroughly tested and verified in the staging environment to minimize downtime and prevent issues. Downtime in production can have a significant impact on revenue and reputation.

❖ Downtime:

Downtime in production can result in revenue loss and negative customer experiences. For example, if a new feature in an e-commerce platform breaks the payment process, it can prevent customers from making purchases, leading to financial losses.

❖ Vulnerabilities:

Cybersecurity is crucial, and updates to software should be checked and verified before deployment. This includes patching and ensuring that software is up-to-date with critical security updates to protect against vulnerabilities.

❖ Reputation:

Downtime or issues in production can damage a company's reputation and erode customer trust. Users may lose confidence in the platform if they encounter frequent problems.