History of Source Code Management

Source Code Management

- Source Code Management (SCM) refers to the practices and tools used to track and control changes in a software project's source code.

1. Early Days (1970s-1980s)

- **Manual Versioning:** Developers managed code versions manually by creating copies of files with different version numbers or dates in the filenames.
- **SCCS (Source Code Control System):** Developed by Bell Labs in the 1970s. It allowed developers to store different versions of source code files and retrieve them as needed.
- **RCS (Revision Control System):** Introduced in the early 1980s. RCS improved upon SCCS by offering better features for branching and merging code.

2. Centralized Version Control Systems (1990s-2000s)

- **CVS (Concurrent Versions System):** Supported concurrent development, allowing multiple developers to work on the same codebase simultaneously.
- **Subversion (SVN):** Successor to CVS. Subversion became popular due to its enhanced features, such as atomic commits, better handling of binary files, and improved support for branching and merging.
- **Perforce:** Popular in large enterprises for speed and scalability.

3. Distributed Version Control Systems (2000s-Present)

- **Git:** Created by Linus Torvalds in 2005, revolutionized SCM.

  - Each developer has a full copy of the repository, allowing for faster operations and more flexible workflows.

  - Git quickly became the dominant version control system, especially in the open-source community .

- **Mercurial:** Provided a distributed approach similar to Git.

4. Modern SCM Practices

- **GitHub, GitLab, Bitbucket:** Platforms built on top of Git to provide additional features like pull requests, issue tracking, and continuous integration/continuous deployment (CI/CD) pipelines.
- **DevOps Integration:** SCM tools integrated into broader DevOps practices. enabling automated testing, deployment, and monitoring as part of the software development lifecycle.

Need for SVC

1.Collaboration

- **Multiple Developers:** Version control allows multiple developers to work on the same project simultaneously without overwriting each other’s changes.
- **Branching:** Developers can create separate branches for different features or fixes, work independently, and later merge their changes back into the main codebase.

2. Tracking Changes

- **History:** Version control systems maintain a complete history of changes made to the codebase, including who made the changes and when.
- **Audit Trail:** This history is invaluable for auditing, understanding the evolution of the project, and tracking down the source of bugs.

3. Reversibility

- **Undo Changes:** If a change introduces a bug or issue, version control allows developers to revert to a previous stable state.
- **Versioning:** Developers can compare different versions of the code, view differences, and revert to any prior version if needed.

4. Backup and Recovery

- **Distributed Repositories:** In distributed version control systems like Git, every developer has a full copy of the project’s history, providing a form of backup.
- **Data Integrity:** Version control ensures that the codebase is preserved, even if individual developers’ systems fail.

5. Consistency Across Environments

- **Releases:** Version control ensures that the same code version is deployed across different environments (development, testing, production), reducing inconsistencies and errors.
- **Automated Deployments:** Integration with CI/CD pipelines ensures that only stable, versioned code is deployed.

6. Branching and Merging

- **Feature Development:** Separate branches allow developers to work on new features without disrupting the main codebase.
- **Safe Integration:** Merging changes back into the main branch can be done safely, with the ability to resolve conflicts if different changes affect the same parts of the code.

7. Collaboration Across Geographical Locations

- **Remote Teams:** Version control systems enable teams spread across different locations to collaborate efficiently by sharing code through a central or distributed repository.

8. Automation and Integration

- **Continuous Integration/Continuous Deployment (CI/CD):** Version control systems are integral to modern CI/CD pipelines, automating testing and deployment processes.
- **Automated Testing:** Every change committed to the version control system can automatically trigger tests, ensuring that new changes do not break the code.

9. Facilitating Code Review

- **Peer Reviews:** Version control systems make it easier to conduct code reviews, as peers can see exactly what changes were made and discuss them before merging into the main branch.
- **Quality Control:** Code reviews help maintain code quality and improve knowledge sharing among team members.

10. Compliance and Legal Protection

- **Regulatory Compliance:** For industries with strict regulatory requirements, version control provides the necessary traceability and documentation.
- **Intellectual Property:** It provides proof of ownership and the evolution of code, which can be critical in legal disputes.