UNIT-III

1Q. What is the importance of source code management in DevOps?

Source Code: Human-readable instructions written by a programmer using a programming language, which are then used to create computer software or applications. Source code contains statements and commands that describe how a program should function, and it is typically stored in text files before being compiled or interpreted into machine-executable code

:The Role of Source Code Management:

Source code management (SCM) refers to the systematic handling, **storage, tracking, and versioning** of all code assets in a **centralized repository**. This practice forms the backbone of a DevOps workflow, bridging developers and operations to ensure team collaboration and seamless automation. In the modern world of software engineering, the importance of SCM cannot be overstated—it ensures code integrity, supports teamwork, and enables the automation necessary for **Continuous Integration (CI) and Continuous Delivery (CD) pipelines**. Tools like **Git, Subversion**, and Mercurial are commonly used SCM systems, but in the DevOps context, the focus is more on how these tools are integrated into the end-to-end workflow rather than which tool itself is used.

Core Reasons Why SCM is Vital in DevOps

1. Centralized Collaboration and Code History

- SCM provides a central code repository that all developers and operations engineers can access and work on simultaneously. It helps to prevent version conflicts—everyone works on the most recent version, and every change is tracked and attributed.
- Having a shared repository is especially crucial when multiple teams are involved. Developers, QA engineers, and operations staff can work concurrently without losing sight of what others are doing, reducing miscommunication and duplication of effort.
- Besides, SCM tools maintain a complete history of changes (commits), making it easy to identify who made what change, when, and why. This is essential for troubleshooting bugs, reviewing features, and even rolling back to previous working versions if necessary.

2. Enabling DevOps Automation

- DevOps places a significant emphasis on automation: building, testing, integrating, and deploying code should be as automatic and smooth as possible.
- Every time code is pushed to the SCM repository, automated build servers (like Jenkins) can trigger CI/CD pipelines. This eliminates manual intervention, speeds up feedback, and increases deployment frequency.

- Source code management acts as the trigger point for automation—once code is committed and pushed, a chain of actions follows, such as compiling the code, running automated tests, and, if successful, moving the code towards deployment.

3. Supports Best Practices: Branching, Merging, and Versioning

- Modern SCM systems allow for multiple branching strategies—feature branches, hotfix branches, and release branches. This allows teams to work on parallel features, quick bug fixes, and stable release versions simultaneously, which is essential in Agile and DevOps environments.
- Merging and pull/request review processes ensure code quality by allowing senior developers or peers to review code before it is integrated into the main branch.
- Tagging releases and following clear versioning conventions (like Semantic Versioning) helps in tracking releases, debugging, and understanding the evolution of the software product over time.

4. Infrastructure as Code

- In DevOps, "everything is code" is a popular axiom. Not just application code, but also infrastructure definitions (e.g., Ansible playbooks, Dockerfiles, Kubernetes descriptors) are stored and managed in SCM repositories.
- This makes infrastructure changes auditable, repeatable, and recoverable, greatly reducing the risks associated with manual configurations. Infrastructure as code is only practical if robust SCM practices exist.

Practical Examples and Benefits

Automation and Continuous Delivery

- When code is committed to the central SCM system, a CI/CD pipeline is automatically triggered. Tools like Jenkins can pull the latest code, run tests, build artifacts, and deploy to test or production environments—all without manual steps.
- This rapid, feedback-driven approach increases productivity and reduces the time-to-market for new features and fixes.

Traceability and Compliance

- Large organizations, especially those in regulated industries, need to prove which change was implemented, who authorized it, and when. SCM logs provide this level of audit and traceability automatically.
- Compliance audits are much easier when all changes, including infrastructure, are versioned and tracked in a single system.

Disaster Recovery and Code Safety

- Mistakes happen in every project. With reliably maintained code history, any broken feature or release can simply be "rolled back" to a previous, stable state—minimizing downtime and risk.
- If a team accidentally deletes code or introduces a critical bug, SCM allows a quick restore of lost or corrupted code.

Distributed and Remote Development

- SCM is indispensable for distributed teams, especially in today's remote/hybrid work environment. Developers spread across cities, countries, or continents can work together seamlessly.
- Using branching, rebasing, and merging, developers can work independently yet integrate their work with minimal friction.

SCM and DevOps Best Practices

- Use a Distributed Version Control System (DVCS): Tools like Git allow developers to work offline, manage their own branches, and later synchronize with the central shared repository.
- Adopt a Standard Branching Strategy: Define guidelines for feature, release, and bug-fix branches to avoid confusion and merge conflicts.
- Code Review and Pull Requests: Integrate review processes to maintain code quality. Pull requests, a popular practice, enable discussion and quality checks before changes are merged.
- Automate Everything Connected to SCM: Builds, tests, deployments, and infrastructure provisioning should all hook into code changes via SCM triggers.
- Document Everything as Code: Not just application and infrastructure, but also documentation, should be version-controlled—making onboarding, knowledge sharing, and updates more systematic..

To quote "Practical DevOps" (Author: Joakim Verona):

"Everything is code... nearly everything can be expressed in codified form, including the applications, the infrastructure, the documentation ... the source code repository is central to the organization. Nearly everything we produce travels through the code repository at some point in its life cycle".

2Q. Describe the evolution of source code control systems. Name a few tools?

Evolution of Source Code Control Systems

In the early days of software development, programmers used to write code individually and store it in files. When projects became bigger and more people started working together, managing the source code became very difficult. Developers faced issues like overwriting each other's work, losing track of versions, and not knowing which changes caused errors. To solve these problems, the idea of **Source Code Control Systems (SCCS)** was introduced.

1. Initial Stage – Local Versioning

The first systems were very simple. Each developer stored code locally and maintained multiple copies with dates or version numbers in file names. This method was not reliable because it caused confusion and wasted storage space. If two developers modified the same file, merging changes was very difficult.

2. Early Source Code Control (1970s–1980s)

The first proper source control tool was **SCCS (Source Code Control System)**, created by AT&T Bell Labs in the 1970s. Soon after, **RCS (Revision Control System)** was introduced in the 1980s. These tools allowed storing versions of files systematically. They used techniques like **delta storage**, where only changes (differences) were saved instead of entire files.

- Advantage: Developers could roll back to previous versions.
- Limitation: These worked mostly for individual developers or small teams and were not suited for distributed development.

3. Centralized Version Control (1990s–2000s)

As projects grew larger, centralized systems were introduced. Examples include:

- **CVS (Concurrent Versions System)**
- **Subversion (SVN)**

In this model, there was one central server where all the code was stored. Developers would "check out" files, make changes, and "commit" them back to the central repository.

- Advantage: Team collaboration improved, and history of code was maintained.
- Limitation: If the central server failed, the whole project was affected. Developers also needed continuous network connectivity.

4. Distributed Version Control (2005 onwards)

To overcome these drawbacks, distributed systems were introduced. The most famous is **Git**, created by Linus Torvalds in 2005 for Linux kernel development. Another example is **Mercurial**.
In these systems, every developer has a full copy of the repository, including complete history. They can commit changes locally and later push or pull changes with others.

- Advantage: Faster performance, offline work possible, strong branching and merging support.
- Git especially became popular because of open-source collaboration and platforms like GitHub and GitLab.

5. Modern Era – Cloud-based Platforms

Today, tools are not only about version control but also about **collaboration, code review, issue tracking, CI/CD integration** etc. GitHub, GitLab, and Bitbucket provide web-based platforms where developers across the world can work together effectively. These tools are at the heart of **DevOps culture** and modern software engineering.

Tools for Source Code Control

Some important tools are:

1. **SCCS (Source Code Control System)** – One of the earliest tools (1970s).
2. **RCS (Revision Control System)** – Improved file-based versioning.
3. **CVS (Concurrent Versions System)** – Popular centralised tool in the 1990s.
4. **SVN (Subversion)** – A widely used centralised tool that improved over CVS.
5. **Git** – Most popular distributed system used today.
6. **Mercurial** – Another distributed system, simple and fast.
7. **Modern Platforms** – GitHub, GitLab, Bitbucket (provide hosting + collaboration).

Conclusion

The evolution of source code control systems reflects the journey of software development itself. From simple file copying to advanced distributed systems, each stage solved the problems of the previous one. Today, tools like Git with platforms such as GitHub are not just version control systems but full collaboration ecosystems. For engineering students, learning Git and GitHub is essential as these are industry standards and form the backbone of DevOps practices.

3Q How does Git support DevOps? Explain Git branching and merging.

In modern software engineering, **DevOps** is the practice that brings together development and operations to deliver software quickly, reliably, and with high quality. The foundation of DevOps is **Source Code Management (SCM)**, and among all SCM tools, **Git** has become the most widely used. Git is a **Distributed Version Control System (DVCS)** created by Linus Torvalds in 2005. It supports DevOps by providing efficient collaboration, version tracking, and integration with automation pipelines.

1. Distributed Nature of Git
Unlike centralized tools like CVS or SVN, Git is distributed. Every developer has a **full copy of the repository with complete history** on their local machine. This allows:

- Developers to commit changes locally without internet connectivity.
- Faster operations like commit, revert, and search.
- Protection against server failures, since copies exist on all machines.

This distributed architecture fits well in DevOps, where **speed and reliability** are key.

2. Parallel and Collaborative Development
DevOps emphasises **team collaboration**. In a project, many developers may work on different features at the same time. Git allows them to do so by using **branches**. Each developer can create their own branch, make changes, and later merge them into the main codebase. This avoids overwriting and conflicts, and increases productivity.

3. Integration with CI/CD Pipelines
Continuous Integration (CI) and Continuous Delivery (CD) are essential parts of DevOps. Git repositories are directly connected with build tools like **Jenkins, GitLab CI, or GitHub Actions**. Whenever a developer pushes new code, the pipeline automatically:

- Builds the code,
- Runs automated tests, and
- Deploys it to staging or production.

This reduces human errors and ensures faster delivery of features.

4. Traceability and Accountability
Every commit in Git contains **metadata** – the author's name, email, timestamp, and commit message. This makes changes fully **traceable**. If a bug appears in production, it is easy to identify who made the change and why. This improves accountability and debugging in a DevOps environment.

5. Flexibility and Tool Ecosystem
Git integrates with modern tools such as **Docker, Kubernetes, Ansible, and Jenkins**. On top of Git, platforms like **GitHub, GitLab, and Bitbucket** provide features like pull

requests, code review, and issue tracking. This makes Git not just a version control tool, but a complete ecosystem that supports the entire DevOps lifecycle.

Git Branching and Merging

Git Branching

Branching is one of the most powerful features of Git. A branch is a **separate line of development** created from the main codebase. It allows developers to try out new ideas, fix bugs, or develop features without disturbing the stable code.

- **Main (or master) branch**: Contains stable and production-ready code.
- **Feature branch**: Used for developing a new feature.
- **Bugfix branch**: Used for fixing specific issues.
- **Release branch**: Prepared for stable releases.

Benefits of Branching:

1. Enables parallel development by multiple teams.
2. Reduces conflicts since developers work in isolation.
3. Supports experimentation without risk to main code.

Branching strategies in DevOps:

- **Feature branching** – A new branch for each feature.
- **Gitflow workflow** – A structured model with master, develop, feature, release, and hotfix branches.
- **Trunk-based development** – Developers work on small, short-lived branches and merge frequently.

Git Merging

Once the development on a branch is complete, the changes are **merged back** into the main branch. Merging combines the work of different developers into a single codebase.

Types of Merging:

1. **Fast-forward merge** – If no other changes were made in the target branch, Git simply moves the pointer forward.
2. **Three-way merge** – When both branches have diverged, Git compares the last common commit and creates a new merge commit.
3. **Rebase (alternative to merge)** – Replays commits of one branch on top of another, giving a cleaner history.

Merge Conflicts

When two developers change the same part of a file, Git cannot decide automatically which change to keep. This situation is called a **merge conflict**. Developers must manually resolve conflicts before completing the merge.

Conclusion

Git supports DevOps by providing a **robust, distributed, and collaborative environment** for managing source code. It integrates smoothly with CI/CD pipelines and ensures faster delivery. Its **branching and merging features** make it possible for teams to work in parallel, experiment safely, and then combine their work into a stable product. For these reasons, Git has become the **industry standard** and an essential tool for engineers working in DevOps culture.

4Q. Differentiate between centralized (CVCS) and distributed version control systems(DVCS).

In software development, **Version Control Systems (VCS)** are essential tools to manage the source code. They keep track of every modification in the code, allow collaboration among developers, and provide rollback to stable versions when needed.

There are two main types of version control systems:

1. **Centralized Version Control Systems (CVCS)**
2. **Distributed Version Control Systems (DVCS)**

Both have the same goal – to manage source code – but their design, workflow, and efficiency are different.

Centralized Version Control Systems (CVCS):

In a centralized model, there is a **single central server** that stores the repository. Developers connect to this server to **check out** files, make modifications, and then **commit** the changes back.

Examples: CVS (Concurrent Versions System), Subversion (SVN), Perforce.

Key Characteristics:

- One central repository, stored on a server.
- Developers must be connected to the server to commit code.
- Code history is maintained only on the central server.
- Admins can easily control access.

Advantages of CVCS:

1. **Simple to learn and use** – Suitable for beginners and small teams.
2. **Single point of control** – Easier administration and backups.
3. **Clear workflow** – Developers know exactly where to commit and retrieve code.

Disadvantages of CVCS:

1. **Single point of failure** – If the server is down, no one can commit or sometimes even access code.
2. **Network dependency** – Continuous internet or LAN connection is required.
3. **Limited collaboration** – Parallel development and branching are difficult compared to DVCS.
4. **Risk of data loss** – If server data is corrupted without backup, full history is lost.

Distributed Version Control Systems (DVCS):

Distributed systems overcome the drawbacks of centralization. In DVCS, every developer has a **full copy of the repository**, including complete history. This allows developers to commit, revert, or view history locally, without depending on a central server.

Examples: Git, Mercurial, Bazaar.

Key Characteristics:

- Each developer's machine is a repository.
- Offline work is possible.
- Developers push and pull changes with others to synchronize.

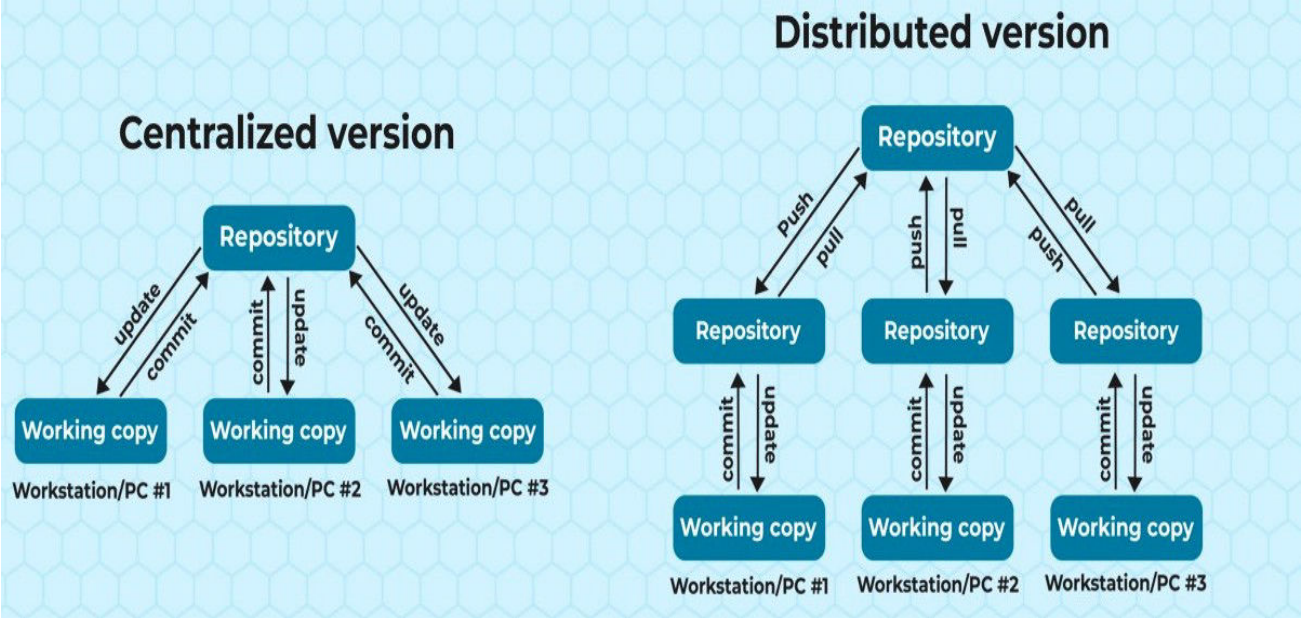
- Supports advanced workflows like feature branching, Gitflow, and pull requests.

Advantages of DVCS:

1. **Offline operations** – Developers can commit, branch, and view history without internet.
2. **High speed** – Local commits and searches are faster.
3. **Data safety** – Since every developer has a full copy, project history is not lost if the central server fails.
4. **Better collaboration** – Supports branching, merging, and parallel workflows.
5. **Flexibility** – Developers can experiment in isolated branches and merge later.

Disadvantages of DVCS:

1. **Learning curve** – Slightly harder to learn for beginners.
2. **Storage requirement** – Each developer's system needs more space as it stores the entire repository.



Comparison Table:

Feature	Centralized VCS (CVCS)	Distributed VCS (DVCS)
Repository	Single central server	Full copy on every developer's PC
Examples	CVS, SVN, Perforce	Git, Mercurial, Bazaar
Network dependency	Always required	Not required for local commits
Speed	Slower (depends on server)	Faster (local operations)
Fault tolerance	Server failure = downtime	Safer, many copies available
Collaboration	Limited	Strong branching & merging
Learning curve	Easier for beginners	Harder at first, but powerful
Backup	Central server backup only	Every developer has a backup

5Q. What is importance to migrate from one SCM system to another(e.g. from SVN to Git)? What are the typical challenges encountered when Migrates source code repositories.

Answer:

SVN (Subversion) is a **Centralized Version Control System (CVCS)** developed by the Apache Software Foundation in 2000. It was designed as an improvement over older systems like CVS (Concurrent Versions System). SVN allows multiple developers to work on the same project by storing all the source code in a **central repository**.
Every developer connects to this repository to **checkout** code, make modifications, and then **commit** changes back. SVN manages the entire history of changes, so teams can track who made which modification, and when.

Importance of Migrating from One SCM System to Another

In the fast-changing software industry, teams sometimes need to **migrate from an older Source Code Management (SCM) tool to a modern one**. For example, many companies have moved from **SVN (Subversion, a centralized system)** to **Git (a distributed system)**.

Reasons for Migration

1. **Better Collaboration**
   - Git allows branching, merging, and pull requests much more efficiently than SVN.
   - Multiple teams across different locations can collaborate smoothly.
2. **Distributed Architecture**
   - In SVN, developers always depend on a central server. If the server fails, work stops.
   - In Git, every developer has a full copy of the repository, which improves reliability.
3. **Performance**
   - Git operations like commit, diff, and history check are very fast because they run locally.
   - SVN operations are slower, as they always contact the central server.
4. **Offline Work**
   - Developers using Git can commit, branch, and view history even without network connectivity.
   - SVN requires a live connection for most operations.
5. **Integration with DevOps Tools**
   - Git integrates easily with modern CI/CD pipelines (Jenkins, GitLab CI, GitHub Actions).
   - Platforms like GitHub, GitLab, and Bitbucket provide project management, code review, and automation features.
6. **Community and Industry Support**
   - Most open-source projects and enterprises have standardised on Git.
   - Moving to Git ensures long-term support, training, and compatibility.

Thus, migration is important to improve team productivity, reliability, and DevOps adoption.

Challenges in Migrating Source Code Repositories

While migration is beneficial, it is not always easy. Teams face both technical and organisational challenges.

1. Repository Size and History

- Large repositories with thousands of commits and branches may take a long time to migrate.
- Preserving **complete history** (all past commits, authors, timestamps) is difficult.

2. Branches and Tags Conversion

- SVN and Git handle branches differently.
- Mapping all SVN branches and tags correctly into Git requires careful planning.

3. Tool and Workflow Differences

- Developers used to SVN commands (checkout, update, commit) must adapt to Git concepts (clone, pull, push, rebase).
- Git introduces new workflows (feature branching, pull requests), which require training.

4. Merge Conflicts and Code Consistency

- During migration, some code may be modified in parallel by teams still working on SVN.
- Ensuring all changes are captured and avoiding data loss is a challenge.

5. Integration with Existing Tools

- Build servers, CI pipelines, and project management tools integrated with SVN may need reconfiguration for Git.
- For example, Jenkins jobs must be updated to fetch from Git instead of SVN.

6. Access Control and Security

- SVN uses a different permission model (path-based access).
- Git platforms use branch-based or repository-level permissions. Mapping users and roles correctly is tricky.

7. Training and Adoption

- Teams must be trained to use Git commands effectively (branching, merging, rebasing).
- Without proper training, mistakes (like force push) can cause disruption.

8. Downtime During Migration

- Migration often requires a **freeze period** where no new commits are allowed until the migration is complete.
- This can affect delivery schedules if not planned properly.

Conclusion

Migrating from one SCM system (like SVN) to another (like Git) is important to achieve **better performance, collaboration, and DevOps integration**. However, it is not just a technical task but also a **cultural change** for the team. The main challenges are repository conversion, workflow differences, tool integration, and developer training. With proper planning and best practices, migration leads to long-term benefits for software teams

6Q. Discuss the pull request model in GitLab with its advantages.

Answer:

In modern software development, especially under **DevOps culture**, collaboration between developers is essential. When multiple developers work on the same project, it is important to merge code in a safe and controlled manner.

In Git-based platforms like **GitLab**, this process is managed using the **Pull Request (PR) model** (also called **Merge Request (MR)** in GitLab). A pull request allows developers to notify others about changes they have made and request those changes to be reviewed and merged into the main branch.

Pull Request Model in GitLab

1. Creating a Feature Branch

- A developer creates a new **branch** from the main branch (e.g., `main` or `develop`) to work on a new feature or bug fix.
- All modifications are committed in this branch.

2. Submitting a Merge Request (Pull Request)

- Once work is complete, the developer pushes the branch to the GitLab repository.
- Then, they open a **Merge Request (MR)** in GitLab.
- This MR acts as a **pull request**, asking reviewers to pull the changes into the target branch.

3. Peer Review and Collaboration

- Other developers and team leads review the code.
- They can add **comments, suggestions, or request changes** line by line.
- GitLab provides an inline discussion system to improve collaboration.

4. Automated Testing and CI/CD Integration

- GitLab integrates with its **CI/CD pipelines**.
- As soon as a pull request is created, automated builds and tests run.
- If tests fail, the MR cannot be merged until issues are fixed.

5. Approval and Merging

- Reviewers provide approvals using GitLab's **approval rules**.
- After getting enough approvals and passing tests, the branch is merged into the target branch.
- The MR is then closed.

Advantages of the Pull Request Model in GitLab

1. Code Quality Improvement

- Every change is reviewed by peers before merging.

- Bugs, logic errors, and violations of coding standards can be caught early.

2. Collaboration and Knowledge Sharing

- Developers learn from each other by reviewing and discussing code.
- Improves teamwork and ensures everyone understands different parts of the project.

3. Safe Integration

- Code from feature branches does not directly go into the main branch.
- Only reviewed and tested code is merged, keeping the main branch stable.

4. CI/CD Automation

- GitLab automatically triggers pipelines for testing, building, and deployment.
- Ensures changes are validated before merging.

5. Transparency and Traceability

- Every MR is stored in GitLab with details such as author, commit history, review comments, and approval status.
- Provides full traceability for auditing and debugging.

6. Supports Branching Strategies

- Works well with workflows like **Gitflow** and **Feature Branching**.
- Multiple developers can work in parallel without conflicts.

7. Security and Permissions

- GitLab allows setting rules like "at least 2 approvals required" or "only senior developers can merge."
- Provides controlled access and secure workflows.

Example Diagram