**Version Control System**

[A. Types of Version Control Systems : 2](#_Toc154054974)

[B. Introduction to Git: 4](#_Toc154054975)

[C. Git, Gitbash and GitHub 5](#_Toc154054976)

[D. Basic Git Commands 7](#_Toc154054977)

[E. Git Branching and Merging 10](#_Toc154054978)

[F. Bitbucket 11](#_Toc154054979)

[G. Working with git in Linux Environment 13](#_Toc154054980)

Version Control System (VCS) and Source Code Management (SCM) are closely related concepts that play a crucial role in software development. They are often used interchangeably, but thereare subtle differences between the two.

**Version Control System (VCS):**

Definition: A version control system is a tool or a set of tools that help in tracking changes to source code over time.

Purpose: The primary purpose of a VCS is to keep track of every modification to the codebase, allowing developers to revert to previous versions, compare changes, and collaborate on a codebase efficiently.

Functionality: VCS records changes to files or sets of files over time. It allows multiple developers to work on a project simultaneously without interfering with each other's work. It provides mechanisms to merge changes from different contributors, handle conflicts, and maintain a complete history of the codebase.

**Source Code Management (SCM):**

Definition: Source Code Management is a broader term that encompasses the management of source code throughout its lifecycle, including version control.

Purpose: SCM involves not only version control but also the organization, storage, retrieval, and distribution of source code. It includes processes and tools for managing code repositories, handling dependencies, and ensuring the integrity of the source code.

Functionality: SCM covers version control but extends to other aspects of code management such as code organization, code reviews, integration with build systems, and deployment processes.

In practical terms, the terms VCS and SCM are often used interchangeably, and many modern VCS tools provide comprehensive SCM features. Examples of popular VCS tools include Git, Mercurial, and Subversion.

*Git is the most widely used VCS today. It is a distributed version control system that allows developers to work offline, commit changes locally, and collaborate with others through remote repositories. Git is known for its speed, flexibility, and powerful branching and merging capabilities.*

*In summary, while VCS primarily focuses on versioning and tracking changes to source code, SCM encompasses a broader set of activities related to the management of source code throughout its lifecycle. Git, as a distributed VCS, is a dominant player in modern software development workflows*.

## Types of Version Control Systems :

There are two main types of Version Control Systems (VCS): centralized version control systems and distributed version control systems.

**Centralized Version Control System (CVCS):**

Description: In a CVCS, there is a central server that stores the complete version history of the project, and clients (developers) interact with this central server to check out the latest version of the code and commit changes.

Advantages:

* Simplicity in terms of setup and use.
* Centralized control makes it easier to manage access permissions.

Disadvantages:

* Single point of failure: If the central server goes down, it can disrupt the workflow for all developers.
* Limited offline capabilities: Developers often need network access to perform version control operations.

Example CVCS: Subversion (SVN) is a popular centralized version control system.

**Distributed Version Control System (DVCS):**

Description: In a DVCS, each developer has a complete copy of the entire repository, including the full version history. Developers can work independently, commit changes locally, and share changes with others through a network.

Advantages:

* Decentralization allows for more flexibility and better support for distributed development.
* Improved offline capabilities: Developers can work offline and commit changes locally before syncing with a central repository.

Disadvantages:

* Initial setup and learning curve can be more complex compared to CVCS.
* Large repositories may consume more storage space on individual machines.

Example DVCS: Git is the most widely used distributed version control system. Mercurial is another example of a DVCS.

Both centralized and distributed version control systems have their pros and cons, and the choice between them often depends on the specific needs and preferences of a development team. In recent years, distributed version control systems like Git have become more prevalent due to their flexibility, efficiency, and support for collaborative, distributed workflows.

One more version control system, A local version control system (VCS) is a version control system that operates on a single machine and manages changes to files and directories locally. Unlike distributed version control systems (DVCS), a local VCS does not involve collaboration with remote repositories or other developers. Instead, it serves as a simple and self-contained solution for tracking changes to files on an individual's computer.

One of the most basic examples of a local VCS is to manually create backups or copies of files with meaningful names or timestamps. However, more sophisticated local version control systems have been developed to provide additional features and benefits. Here's a simple illustration:

Example: RCS (Revision Control System)

RCS is one of the early local version control systems designed to manage revisions of individual files. Developers use commands like ci (check-in) to commit changes and create a new revision. Commands like co (check-out) retrieve a specific revision of a file for editing.

RCS maintains a file called "RCS file" for each versioned file, storing revision information.

While RCS served as a local VCS, it had limitations when it came to collaboration and handling larger projects. Modern distributed version control systems like Git and Mercurial have largely replaced local VCS solutions for many development scenarios due to their distributed and collaborative nature.

## Introduction to Git:

Git is a distributed version control system (DVCS) for tracking changes in source code during software development. It is designed to handle everything from small to very large projects with speed and efficiency.

Git was created by Linus Torvalds in 2005 for the development of the Linux kernel. It has since become the most popular DVCS in the world, and is used by a wide range of projects, including:

* Web development
* Mobile app development
* Game development
* System administration
* Scientific computing

**Key features of Git include:**

* **Distributed:** Every developer has a complete copy of the project history on their own machine. This makes it easy to collaborate with other developers, even if they are not connected to the internet.
* **Speed:** Git is very fast, even for large projects. This is because it only tracks changes to files, rather than the entire file history.
* **Efficiency:** Git is very efficient with storage space. This is because it uses a technique called delta compression, which only stores the differences between different versions of a file.

**Git terminology:**

* **Repository:** A repository is a collection of files and directories that are tracked by Git.
* **Commit:** A commit is a snapshot of the repository at a particular point in time.
* **Branch:** A branch is a line of development within a repository.
* **Merge:** A merge is the process of combining two or more branches into a single branch.
* **Remote:** A remote is a reference to another Git repository.
* **Pull:** A pull is the process of fetching and merging changes from a remote repository.
* **Push:** A push is the process of sending your changes to a remote repository.

**Benefits of using Git:**

Here are some of the reasons why we need Git:

* **Track changes:** Git makes it easy to track changes to your code over time. This can be helpful for debugging, collaboration, and retrospectives.
* **Revert changes:** If you make a mistake, you can easily revert to a previous version of your code.
* **Collaborate with others:** Git makes it easy to collaborate with other developers on the same project.
* **Branch and merge:** Git's branching and merging features make it easy to work on different features or bug fixes at the same time.
* **Share code:** Git makes it easy to share your code with others by pushing it to a remote repository, such as GitHub or GitLab.
* **Non-linear development:** Git's branching model allows for non-linear development, where developers can work on different features or fixes without interfering with each other's work. This can be especially helpful for large projects with multiple contributors.
* **Offline collaboration:** Git allows developers to work offline and then sync their changes later, making it a useful tool for distributed teams.
* **Security:** Git repositories can be made private to protect sensitive information.
* **Auditing:** Git provides a complete history of changes to a codebase, making it easy to audit changes and identify who made them and when.
* **Reproducibility:** Git allows developers to easily reproduce any previous state of a codebase, making it easier to debug problems and roll back changes if necessary.
* **Integration with other tools:** Git can be integrated with other development tools, such as IDEs and build systems, to streamline the development workflow.

In addition to these benefits, Git is also a free and open-source tool, which means that it is available to anyone to use and modify. This has made it the most popular DVCS in the world, and it is used by a wide range of projects, from small personal projects to large enterprise software applications.

Overall, Git is a powerful and versatile tool that can make any software development project more efficient and successful.

**Getting started with Git:**

1. Install Git on your computer. You can download Git from the official website: https://git-scm.com/downloads
2. Create a Git repository for your project.
3. Add files to your Git repository.
4. Make changes to your files.
5. Commit your changes to your Git repository.
6. Push your changes to a remote repository.

The official Git documentation is a great place to start: https://git-scm.com/doc There are also many online tutorials and courses available.

## Git, Gitbash and GitHub

**Git** is a distributed version control system (DVCS) for tracking changes in source code during software development. It is designed to handle everything from small to very large projects with speed and efficiency.

**Git Bash** is a free and open-source command-line environment and graphical user interface (GUI) for the Git version control system. It is available for Windows, macOS, and Linux.

**GitHub** is a web-based hosting service for Git repositories. It provides a platform for developers to store, manage, and collaborate on their code. GitHub also provides a number of social features, such as issue tracking and pull requests, that make it easy for developers to work together.

Git is a tool for tracking changes in code, while Git Bash is a tool for interacting with Git from the command line. GitHub is a platform for hosting and collaborating on Git repositories.

**The key differences between Git and Git Bash:**

Git Bash is a valuable tool for developers who use Git. It makes it easy to interact with Git from the command line, and it provides a graphical interface for those who prefer to use a GUI.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Git** | **Git Bash** |
| Type of tool | Distributed version control system (DVCS) | Command-line environment and GUI for Git |
| Purpose | Track changes in source code | Interact with Git from the command line |
| Platform | Windows, macOS, Linux | Windows, macOS, Linux |
| Installation | Requires installation | Included with Git for Windows |
| Usage | Requires knowledge of Git commands | Easy to use for beginners |

Git and Git Bash are two different tools related to version control, but they serve different purposes.

**The following is an example of how Git and Git Bash are used together:**

1. A developer makes a change to a file in a Git repository.
2. The developer uses Git Bash to stage the change for commit.
3. The developer uses Git Bash to commit the change.
4. The developer uses Git Bash to push the change to a remote repository.

If you are a developer who uses Windows, you will need to use Git Bash to use Git. However, if you use a different operating system, you can use Git without Git Bash.

Here are some additional things to keep in mind about Git and Git Bash:

* Git Bash is not required to use Git. You can use Git on any operating system without Git Bash.
* Git Bash is a free and open-source tool.
* Git Bash is available for download from the Git website: <https://git-scm.com/downloads>

**Key differences between git and github**

Git and GitHub are two closely related tools that are often used together to manage software projects effectively. However, they have different purposes and are used in different ways.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Git** | **GitHub** |
| Type of tool | Distributed version control system (DVCS) | Web-based hosting service for Git repositories |
| Purpose | Track changes in source code | Store, manage, and collaborate on Git repositories |
| Platform | Windows, macOS, Linux | Web-based |
| Installation | Requires installation | No installation required |
| Usage | Requires knowledge of Git commands | Easy to use for beginners |
| Cost | Free | Free and paid plans |
| Social features | None | Issue tracking, pull requests, social coding |

**Here is an example of how Git and GitHub are used together:**

1. A developer creates a new Git repository for their project.
2. The developer adds files to their Git repository.
3. The developer makes changes to their files.
4. The developer commits their changes to their Git repository.
5. The developer clones their Git repository to GitHub.
6. The developer pushes their changes to their GitHub repository.
7. Other developers can clone the GitHub repository and pull the latest changes.

## Basic Git Commands

1. git init will create a new local GIT repository. The following Git command will create a repository in the current directory:

**git init**

*Alternatively, you can create a repository within a new directory by specifying the project name.*

1. git clone is used to copy a repository. If the repository lies on a remote server, use:

**git clone username@host:/path/to/repository**

Conversely, run the following basic command to copy a local repository:

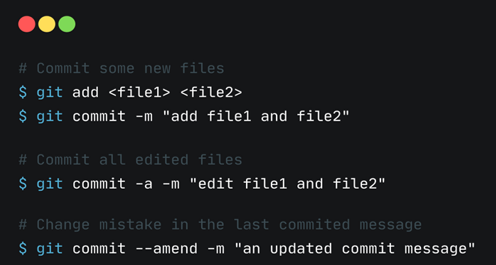
**git clone /path/to/repository**

1. git add is used to add files to the staging area. For example, the basic Git following command will index the temp.txt file:

**git add <temp.txt>**

1. git commit will create a snapshot of the changes and save it to the git directory.

**git commit –m “Message to go with the commit here”**



1. git config can be used to set user-specific configuration values like email, username, file format, and so on. To illustrate, the command for setting up an email will look like this:

**git config --global user.email** [**youremail@example.com**](mailto:youremail@example.com)

1. The –global flag tells GIT that you’re going to use that email for all local repositories. If you want to use different emails for different repositories, use the command below:

**git config --local user.email** [**youremail@example.com**](mailto:youremail@example.com)

1. git status displays the list of changed files together with the files that are yet to be staged or committed.

**git status**

1. git push is used to send local commits to the master branch of the remote repository. Here’s the basic code structure:

**git push origin <master>**

1. git checkout creates branches and helps you to navigate between them. For example, the following basic command creates a new branch and automatically switches you to it:

*To switch from one branch to another, simply use:*

**git checkout <branch-name>**

1. git remote lets you view all remote repositories. The following command will list all connections along with their URLs:

**git remote –v**

1. To connect the local repository to a remote server, use the command below:

**git remote add origin <host-or-remoteURL>**

1. Meanwhile, the following command will delete a connection to a specified remote repository:

**git remote rm <name-of-the-repository>**

1. git branch will list, create, or delete branches. For instance, if you want to list all the branches present in the repository, the command should look like this:

**git branch**

1. If you want to delete a branch, use:

**git branch –d <branch-name>**

1. git pull merges all the changes present in the remote repository to the local working directory.

**git pull**

1. git merge is used to merge a branch into the active one.

**git merge <branch-name>**

1. git diff lists down conflicts. In order to view conflicts against the base file, use

**git diff --base <file-name>**

1. The following basic command is used to view the conflicts between branches before merging them:

**git diff <source-branch> <target-branch>**

1. To list down all the present conflicts, use:

**git diff**

1. git tag marks specific commits. Developers usually use it to mark release points like v1.0 and v2.0.

**git tag <insert-commitID-here>**

1. git log is used to see the repository’s history by listing certain commit’s details. Running the command will get you an output that looks like this:

**git log**

1. git reset command will reset the index and the working directory to the last git commit’s state.

**git reset --hard HEAD**

1. git rm can be used to remove files from the index and the working directory.

**git rm filename.txt**

1. git stash command will temporarily save the changes that are not ready to be committed. That way, you can go back to that project later on.

**git stash**

1. git show is a command used to view information about any git object.

**git show**

1. git fetch allows users to fetch all objects from the remote repository that don’t currently reside in the local working directory.

**git fetch origin**

1. git ls-tree allows you to view a tree object along with the name, the mode of each item, and the blob’s SHA-1 value. Let’s say you want to see the HEAD, use:

**git ls-tree HEAD**

1. git cat-file is used to view the type and the size information of a repository object. Use the -p option along with the object’s SHA-1 value to view the information of a specific object, for example:

**git cat-file –p d670460b4b4aece5915caf5c68d12f560a9fe3e4**

1. git grep lets users search through committed trees, working directory, and staging area for specific phrases and words. To search for [www.hostinger.com](http://www.hostinger.com/) in all files, use:

**git grep** [**"www.hostinger.com"**](http://www.hostinger.com/)

1. gitk shows the graphical interface for a local repository. Simply run:

**gitk**

1. git instaweb allows you to browse your local repository in the git-web interface. For instance:

**git instaweb –httpd=webrick**

1. git gc will clean unnecessary files and optimize the local repository.

**Git gc**

1. git archive lets users create a zip or a tar file containing the constituents of a single repository tree. For instance

**git archive --format=tar master**

1. git fsck performs an integrity check of the git file system and identifies any corrupted objects.

**git fsck**

1. git rebase is used to apply certain changes from one branch to another. For instance:

**git rebase master**

1. git fork creates a copy of the original repository on your GitHub account

## Git Branching and Merging

In Git, branching and merging are fundamental concepts that enable developers to work on separate features or fixes in parallel and later integrate those changes back into the main codebase. Here's an overview of Git branching and merging:

**Branching in Git:**

Create a Branch: Developers create branches to work on a specific feature, bug fix, or task without affecting the main codebase. The main branch, often named "master" or "main," represents the stable version of the code.

***git branch feature-branch***

Switch to a Branch: Developers switch to the branch where they want to work.

***git checkout feature-branch***

In Git versions 2.23 and later, you can use git switch:

***git switch feature-branch***

Or, combine branch creation and switch:

***git switch -c feature-branch***

Commit Changes: Developers make changes and commit them to the branch.

***git commit -m "Implemented feature X"***

**Merging in Git:**

Merge Changes: Once the work on a feature is complete, the changes are merged back into the main branch.

# Switch to the main branch

***git checkout main***

# Merge changes from the feature branch

***git merge feature-branch***

Alternatively, you can use a single command to switch and merge:

***git switch main***

***git merge feature-branch***

Merging is a common operation in Git, and understanding how to handle conflicts and execute merges efficiently is essential for smooth collaboration in a version-controlled environment.

*Fast-forward Merge:*

*If there are no new changes in the target branch since you created the source branch, Git may perform a "fast-forward" merge, meaning it simply moves the pointer of the target branch to the latest commit in the source branch using rebase.*

*Merge Strategies:*

*Git provides different merge strategies, such as recursive and octopus. In most cases, the default recursive strategy works well. However, for more complex scenarios, you may explore other strategies.*

*Rebasing in Git is an alternative to merging that allows you to integrate changes from one branch into another by moving or combining a sequence of commits. Unlike merging, which creates a new commit to represent the merge, rebasing rewrites the commit history. Rebasing is particularly useful for creating a linear and cleaner commit history.*

**Resolve Conflicts:** If there are conflicting changes between branches, Git will prompt you to resolve the conflicts manually. After resolving conflicts, you need to commit the changes.

***git commit -m "Merge branch 'feature-branch' into main"***

Delete the Branch: After merging, you can delete the feature branch if it's no longer needed.

***git branch -d feature-branch***

Use -d to delete if the changes are already merged. Use -D if you want to force delete, even if there are unmerged changes.

**Pull Requests (GitHub/GitLab/Bitbucket):**

In collaborative environments, developers often use pull requests (PRs) to propose changes. A pull request is a request to merge changes from one branch into another. It allows for code review, discussion, and collaboration before merging.

# Assuming you are on the feature branch

***git push origin feature-branch***

Then, on the repository hosting service (e.g., GitHub, GitLab, Bitbucket), you can create a pull request to merge changes from feature-branch into the main branch.

These processes make Git a powerful and flexible version control system, enabling collaborative development with multiple contributors working on different features concurrently. In Git, branching and merging are fundamental concepts that enable developers to work on separate features or fixes in parallel and later integrate those changes back into the main codebase. Here's an overview of Git branching and merging:

## Bitbucket

Bitbucket is a popular web-based platform designed to facilitate collaboration and version control for software development projects. With support for both Git and Mercurial, Bitbucket caters to a wide range of development workflows.

**Version Control with Bitbucket**

* Bitbucket allows users to create repositories to host their source code. Developers can initialize repositories for new projects or import existing ones.
* Bitbucket supports branching and merging, essential for collaborative development. Teams can create branches for features or bug fixes, and later merge changes back into the main branch.
* Pull Requests (PRs) enable developers to propose changes, discuss modifications, and collaborate before merging code. Bitbucket's PR system facilitates code review and ensures code quality.

## Working with git in Linux Environment

1. Lunch Linux EC2 instance in AWS account
   1. Create AWS free account using your credit/ Debit card.
   2. Login to your account
   3. Set region as [ap-south-1](https://ap-south-1.console.aws.amazon.com/ec2/home?region=ap-south-1#InstanceDetails:instanceId=i-092d762feaa33881b)



A screenshot of a computer

Description automatically generated

* 1. Goto EC2 service

A screenshot of a computer

Description automatically generated

* 1. Click on Launch Instance

A screenshot of a computer

Description automatically generated

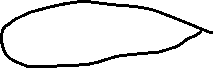


* 1. Follow below steps to launch Linux image

Enter any name ex: Linux\_Server

A screenshot of a computer

Description automatically generated



Select Linux image

A screenshot of a computer

Description automatically generated



Select proceed with no key value pair

A screenshot of a computer

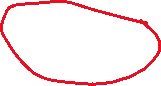
Description automatically generated



Keep all remaining with default settings and click on Lauch instance

A screenshot of a computer

Description automatically generated



It takes few minutes to launch instance to make it stats to running state

A screenshot of a computer

Description automatically generated



Select instance and click on connect

A screenshot of a computer

Description automatically generated



Click on connect

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

1. Run the following commands to install git

Sudo su -

Yum update

Yum install git

1. Run following Git config commands

**git config --global user.email** [**youremail@example.com**](mailto:youremail@example.com)

**git config --global user.email** [**youremail@example.com**](mailto:youremail@example.com)

1. Establish a secure connection between **Git** and **Gitbash** using SSH key.
2. Start a git project.
3. Work with git commands