* **Stash -** Git's stash feature enables developers to save changes temporarily without committing them. This is useful when working on one task but needing to switch to another without committing incomplete changes. Stashing allows for clean and organized workspaces without losing any modification
* **Git Blame -** The git blame command helps developers identify the author and last modification details for each line of code in a file. This feature is valuable for understanding the origin of changes, tracing bugs, and facilitating accountability and collaboration within a team.
* **Git Rebase -** Git rebase allows developers to modify the commit history, altering the order, combining or splitting commits, and resolving conflicts. It provides a flexible way to tidy up the commit history and prepare clean, linear changesets for integration.
* **Cherry Picking -** With Git's cherry-pick feature, developers can select specific commits from one branch and apply them to another. This is useful when you want to incorporate specific changes or fixes from one branch into another without merging the entire branch.

**3 Stage Architecture in Git**

**1. Working Directory**

The working directory is where developers make modifications to their project's files. These changes are not yet tracked by Git and exist outside of its version control system. The working directory represents the developer's local workspace, and it is in this stage that they edit files and prepare updates.

**2. Staging Area (Index)**

The staging area is an intermediary step between the working directory and the repository. When a developer wants to commit their changes, they must first add the modified files to the staging area. In this stage, files are prepped and marked for inclusion in the next commit. The staging area acts as a snapshot of the changes about to be committed.

**3. Repository**

The repository is where all the committed changes are stored permanently. When a developer commits changes, Git creates a snapshot of the files in the staging area and stores it as a new commit in the repository. The repository maintains a complete history of the project, enabling developers to traverse back in time, view previous states, and compare different versions.

* **Git add** is the command that allows you to promote changes from the working directory to the staging area.
* **Git Commit** - Once your **changes** are staged, you can commit them to your repository using the git commit command.
* **git add -i** - to start the interactive staging session.
* **git commit --amend -** This command lets you modify the last commit by adding more changes or changing the commit message.
* **git clone -** creates a local copy of a remote repository
* **Git status** - displays current state of repo including modified and staged files.
* **Git push** - uploads local commits to remote repo
* **Git pull** - It fetches changes from the remote repository and merges them into the current branch of your local repository.
* **.gitignore** file is a plain text file where each line contains a pattern for files/directories to ignore. It's typically placed at the root of the repository, although it can be placed in any directory in the repository. The patterns listed in the file apply recursively to the directory that the .gitignore file is in.
* **Git fetch** - Downloads new data from remote repo but doesn't integrate it into working files.
* **git branch -r** or **git branch -a :** to view all remote branches that git is tracking
* **git mv newname oldname :** rename file and stage renaming operation

**Branching**

Independent line of development for the project. The default branch in Git is the master branch. When you commit changes without specifying a branch, they get added to the master branch.

**Adding & Committing files to staging area**

* git add --all
* git add -A
* git add .
* Git commit -m "message"
* Git log - view repo history

**Best Practices for using GIT**

1. Commit Early and Often

2. Write Meaningful Commit Messages

3. Use Branches

4. Keep Your Branches Up to Date

5. Avoid Altering Published History

6. Use gitignore

7. Perform Regular Housekeeping

**Git Submodules**

* Repositories nested inside parent repositories.
* Allow to incorporate and track version history of external projects or libraries directly into your project.
* Highly useful when you want to share code that you also need to work on independently of the parent project, or when you want to include third-party code that is developed and maintained elsewhere.

**Adding Submodule**   
  
Use the git submodule add command, followed by the URL of the repository you want to include. This command creates a new file called .gitmodules, which maps the project URL to your local subdirectory.

**git submodule add** [**https://github.com/user/repo.git**](https://github.com/user/repo.git) **your-submodule-name**  
  
To pull down changes from the submodule repositories to your local repository, you use the git submodule update command.

**git submodule update --remote**

**Git Hooks**

Git Hooks are scripts that Git executes before or after events such as commit, push, and receive. They are used for automating tasks in the development workflow, from enforcing code policies, adjusting your workflow, to triggering builds in continuous integration.

These scripts are stored in the 'hooks' directory within the '.git' directory of a Git repository and are not transferred with the rest of your repository when you clone or push it.

**Git Stash**

Git Stash acts like a clipboard, where you can store changes that you don't want to commit immediately, but also don't want to lose. These could be half-done features, experimental code, or quick bug fixes. The stash is a temporary storage where you can keep these changes, switch contexts, and retrieve them when you're ready.

Used in scenarios of :

* **Switching Branches:** If you are in the middle of a task on one branch and need to switch to a different branch for a while, you can stash your changes and retrieve them later.

* **Pulling Updates:** If you have uncommitted changes and need to pull updates from the remote repository, you can stash your changes, pull the updates, and then reapply your changes.

* **Cleaning Up:** If your workspace is cluttered with experimental changes, you can stash them to clear your workspace.

**Stashing Changes:** To stash changes, use git stash save "message". Replace "message" with a description of what you're stashing.

**git stash save "work in progress for feature X"**

**Listing Stashes:** To see what stashes you currently have, use git stash list.

**git stash list**

**Applying Stashed Changes:** To apply a stashed change, use git stash apply. This will apply the most recently stashed changes. If you want to apply a specific stash, use git stash apply stash@{n}, where 'n' is the stash number.

**git stash apply**

**Git Cherry Pick**

Select a single commit from one branch and apply it onto another.

Used in:

**Isolating Specific Changes:** If you want to apply specific changes without merging an entire branch, cherry-picking allows you to do this cleanly.

**Bug Fixes:** If you've fixed a bug in one branch and want to apply the fix to other branches, cherry-pick is a straightforward way to achieve this.

**Collaborative Work:** When multiple people are working on a project, cherry-picking allows you to bring in useful commits from other developer's branches.

**git cherry-pick <commit-hash>**

**Reflog**

* Mechanism to record when the tips of branches and other references were updated in the local repository.
* Log of where your HEAD and branch references have been.
* Provides a way to access commits that are no longer in your current commit history.

**Use:**

* **Undoing Rebase:** If you've performed a rebase and something went wrong, or you simply want to go back to the state before the rebase, git reflog can help you locate the commit you need.
* **Recovering Lost Commits:** If a branch was deleted, or some other action resulted in lost commits, git reflog allows you to find these 'lost' commits.
* **Navigating Git History:** The reflog is a way to see all actions (commits, checkouts, merges, etc.) you've done in a repository, providing an overview of your Git history.

**Cloning**

* Process of creating a local copy of a remote repository.
* Copies all versions of all files in the repository, allowing you to work on the project from your local machine while still being able to push your changes to the remote repository.
* This is particularly useful when you're working on a project with multiple contributors or when you want to work on a project from different machines.

**Forking**

* Fork is a copy of a repository that allows you to experiment and make changes without affecting the original project.
* Forking a repository is an essential part of contributing to open source projects, as it allows you to freely make changes to project files, and propose those changes back to the original project (known as a pull request).

**Pull Requests**

Pull requests are GitHub's mechanism for proposing changes from a forked repository to the original project. When you create a pull request, you are essentially requesting the project maintainers to review and merge your modifications into the main codebase.

**Branches**

Branches in GitHub are independent copies of a repository that allow for parallel development. They enable multiple individuals to work on different features or bug fixes simultaneously without affecting the main codebase.

**Fetch + Merge = Pull**

**Fetch**

Git fetch is used to update the local repository with the latest changes from a remote repository without automatically merging the changes into the current branch. The command retrieves the latest commits, branches, and tags from the remote repository and stores them in the local repository, allowing you to review and merge the changes at a later stage.

Git fetch is a non-destructive operation, meaning it doesn't modify your working directory or make any changes to your local branch. It only updates the remote tracking branches in your local repository.

**Pull**

Git pull combines the functionality of git fetch and git merge into a single command. It updates the local repository with the latest changes from the remote repository and automatically merges those changes into the current branch. Git pull effectively fetches the changes and merges them in one step.

Git pull is a convenient command when you want to quickly update your local repository and incorporate the remote changes into your working directory.

**1. Git Fetch is useful when:**

- You want to see the latest changes without merging them immediately.

- You want to review the changes before merging.

- You have multiple branches and want to update the remote tracking branches.

**2. Git Pull is useful when:**

- You want to quickly update your local repository and merge the changes.

- You are confident that you don't need to review the changes before merging.

- You are working on a single branch and want to simplify the update process.